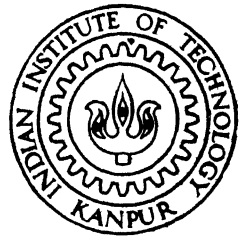
Parallel Generation of Permutations, Combinations and Derangements

by

D T V Rama Krishna Rao



Department of Computer Science & Engineering

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

May, 1997

Parallel Generation of Permutations, Combinations and Derangements

A Thesis Submitted
in Partial Fulfillment of the Requirements
for the Degree of
Master of Technology

by

D T V Rama Krishna Rao

to the

Department of Computer Science & Engineering
Indian Institute of Technology, Kanpur

May, 1997

CSE-1997-M-RAO-PAR

Certificate

Certified that the work contained in the thesis entitled "Parallel Generation of Permutations, Combinations and Derangements", by Mr.*D T V Rama Krishna Rao*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

S. Saxena 6/5/97

Dr. Sanjeev Saxena,
Assosciate Professor,
Dept. of CSE,
IIT Kanpur.

May, 1997

Acknowledgments

I am deeply indebted to my thesis supervisor Dr.Sanjeev Saxena for guiding me. He has been an immense source of inspiration. His insight and his approach of "looking beyond the present approach" have helped in i a number of ways to improve the work presented in this thesis. Sir, I am grateful to you.

My internal and external examiners Dr. R.K Ghosh and Dr. Rajiv Varma have made helpful comments. I am thankful to them.

It would be very difficult to imagine the world without friends. Praveen, Srinivas (kommu), Raghuram, D Srinivas, Samir Shaw, Samir Goel and Kaladhar have provided the needed impetus and "caustic criticism" which provoked me to do my work in a vigorous manner. Friends, I thank you.

Much of my work has been done in the cosy surroundings of iitk. It is simply superb. The trees, the peacocks, the birds have provided the needed enthusiasm to sustain my work. They provided the companionship during my evening strolls. Dear friends, I love you.

Affectionately
dedicated
to the flora and fauna at iitk.

Abstract

We present new and efficient parallel algorithms for generation of combinatorial objects. The combinatorial objects we generate include permutations, combinations and derangements. Our algorithms can be categorized based on time into two types: linear time and poly-logarithmic. The tool that enables the linear time algorithms is the concept of *Combinatorial tree*. The poly-logarithmic time algorithms are derived using *divide and conquer* paradigm. In general our algorithms can be looked at as giving *algorithmic interpretation* to well known combinatorial identities. The abstract models of parallel computation used are members of the Parallel random access machine (PRAM) family. The results are summarized below, grouped according to problems.

Permutations: New efficient algorithms for generation of all r-permutations of n distinct objects are presented. We present four basic algorithms for the generation of permutations. The first algorithm is cost optimal for generating r-permutations in lexicographic order with time complexity $O(r)$ on CREW PRAM. It is based on building a *permutation tree*. This algorithm is marked for its simplicity when compared to the existing algorithms. The second algorithm generates all the r-permutations of n objects with a time complexity of $O(\log n)$ and a cost of $O(r \log r P(n,r))$ on EREW PRAM. This algorithm also runs with a time of $O(\log n)$ and work of $O(r \log\log r P(n,r))$ on ARBITRARY CRCW PRAM. An extension of this algorithm for lexicographic generation of permutations runs with time $O(\log n)$ and work of $O(r \log r P(n,r))$ but on CREW PRAM. We also discuss the randomized and non-conservative variants of this algorithm; these variations are optimal and run with a time complexity of $O(\log n)$. We later obtained an optimal version of this algorithm which generates r-permutations in lexicographic order on EREW PRAM in $O(\log n)$ time with $O(rP(n,r))$ work. The third algorithm generates r-permutations lexicographically with a time complexity of $O(\log r(\log r + \log\log * n))$ with optimal work $O(rP(n,r))$ on CREW PRAM. This algorithm can also be implemented to work with a time complexity of $O(\log r(\log r + \alpha(n)))$ with optimal work on COMMON CRCW PRAM. The fourth algorithm runs with a time of $O(\log n)$ with optimal work on CREW PRAM. The last two algorithms are designed based on divide-and-conquer paradigm.

Combinations: We present three optimal algorithms for the lexicographic generation of r-combinations of n distinct objects. The first algorithm runs with a time complexity

of $O(r)$ and optimal work $O(rC(n,r))$ on CREW PRAM and is based on building a *combination tree*. The second algorithm has a time complexity of $O(\log r(\log r + \log\log * n))$ on CREW PRAM and $O(\log r(\log r + \alpha(n))$ on COMMON CRCW PRAM and is based on divide and conquer on the parameter r. The third algorithm runs with a time of $O(\log n)$ on CREW PRAM and is based on divide and conquer on the parameter n.

Derangements: We present new, efficient and optimal parallel algorithms for the generation of derangements. The first algorithm runs in $O(n)$ time with optimal work $O(nD(n))$ on CREW PRAM. This is based on building a *derangement tree*. We present a novel method to show that given an $O(T)$ time and $O(W)$ work algorithm for lexicographic generation of all permutations of n objects, we can automatically generate all derangements of n objects in $O(T + \log n)$ time with $O(W)$ work on the same model on which permutation generation algorithm runs. This results in an $O(\log n)$ time $O(nD(n))$ optimal work algorithm for lexicographic generation of derangements on EREW PRAM. These improvements are a result of a ranking function, designed for derangement. The ranking function is also of independent interest. We discuss parallelization of ranking function on EREW and CREW PRAMs.

Contents

Chapter 1

Introduction

In recent years there has been tremendous effort on the part of researchers throughout the world to develop fast and efficient parallel algorithms. The reason for this urgency is, that sequential machines are reaching their limits in terms of their computing power. It appears parallel model of solving problems is the only practical approach to meet the computing needs of the coming generation. In accordance with this idea, parallel computers have been built. More over, the parallel computers are becoming more and more feasible and lack of efficient parallel algorithms may become a bottleneck for their commercial use. The work described in this thesis conforms with the general goal of developing fast and efficient algorithms. We propose new efficient parallel algorithms for generation of combinatorial objects.

1.1 Combinatorial Generation

The systematic generation of combinatorial objects is a fundamental problem in combinatorial computation. The enumeration of combinatorial objects occupies an important place in computer science due to its many applications in science and engineering[3, 30, 31]. They have applications in the design of other algorithms like subset-sum, knapsack and base-enumeration problems[15]. In this thesis we present parallel algorithms for the generation of r-permutations, r-combinations and derangements for distinct objects. We assume those distinct objects to be the first n positive integers.

We begin with some definitions. Let S be a set consisting of n distinct items, say,

the first n positive integers. So, $S = \{1, 2, \ldots, n\}$. For our purposes, a combinatorial object can be considered as an ordered selection of some integers out of S.

Now, let $x = (x_1 x_2 \ldots x_r)$ and $y = (y_1 y_2 \ldots y_r)$ be two combinatorial objects of S. We say that x *precedes* y *in lexicographic order* if either $x_1 < y_1$ or if there exists an integer i, $1 \leq i \leq r$, such that $x_j = y_j$ for all $j < i$ and $x_i < y_i$.

Based on the above definition, combinatorial objects can be ordered in lexicographic order. So, each combinatorial object of a given length has an associated index in lexicographic order. By *ranking* we mean getting this index of an arbitrary combinatorial object. Similarly, *unranking* means getting the combinatorial object given its index in lexicographic order. So, unranking is the inverse operation of ranking.

Permutations

An r-*permutation* of S is obtained by selecting r distinct integers out of S and arranging them in some order. Thus, for example, for $n = 10$ and $r = 4$, a 4-permutation might be (5 7 9 3). Two r-permutations are different (or *distinct*) if they differ either with respect to the items they contain or with respect to the order of the items. The number of distinct r-permutations of n items is denoted by $P(n,r)$, where $P(n,r) = \frac{n!}{(n-r)!}$. Thus, for $n = 4$, there are twenty four distinct 3-permutations. In the special case, where $r = n$, $P(n,n) = n!$.

The 3-permutations of $\{1,2,3,4\}$ in lexicographic order are shown below:

(1 2 3), (1 2 4), (1 3 2), (1 3 4),
(1 4 2), (1 4 3), (2 1 3), (2 1 4),
(2 3 1), (2 3 4), (2 4 1), (2 4 3),
(3 1 2), (3 1 4), (3 2 1), (3 2 4),
(3 4 1), (3 4 2), (4 1 2), (4 1 3),
(4 2 1), (4 2 3), (4 3 1), (4 3 2).

Since there are $P(n,r)$ r-permutations, each of length r, permutation generation has a trivial lower bound of $\Omega(P(n,r)r)$.

The current best parallel algorithm for the generation of r-permutations runs with a time complexity of $O(r)$ and is cost optimal[34]. This algorithm does not produce the permutations in lexicographic order.

We present new efficient parallel algorithms for generation of all r-permutations of n distinct objects. We present four basic algorithms for the generation of permutations in

Chapters 3, 4 and 5. The first algorithm is cost optimal for generating r-permutations in lexicographic order with time complexity $O(r)$ on CREW PRAM. It is based on building a *permutation tree*. This algorithm is marked for its simplicity when compared to the existing algorithms. The second algorithm generates all the r-permutations of n objects with a time complexity of $O(\log n)$ and a cost of $O(P(n,r)r\log r)$ on EREW PRAM. This algorithm also runs with a time of $O(\log n)$ and work of $O(P(n,r)r\log\log r)$ on ARBITRARY CRCW PRAM. An extension of this algorithm for lexicographic generation of permutations runs with time $O(\log n)$ and work of $O(P(n,r)r\log r)$ but on CREW PRAM. We also discuss the randomized and non-conservative variants of this algorithm; these variations are optimal and run with a time complexity of $O(\log n)$. We later discuss an optimal version of this algorithm which generates r-permutations in lexicographic order in $O(\log n)$ time with $O(rP(n,r))$ work on EREW PRAM. The third algorithm generates r-permutations lexicographically with a time complexity of $O(\log r(\log r + \log\log * n))$ with optimal work $O(P(n,r)r)$ on CREW PRAM. This algorithm can also be implemented to work with a time complexity of $O(\log r(\log r + \alpha(n)))$ with optimal work on COMMON CRCW PRAM. The fourth algorithm runs with a time of $O(\log n)$ with optimal work on CREW PRAM. The last two algorithms are designed based on divide-and-conquer paradigm.

Combinations

An r-combination of S is obtained by selecting r distinct integers out of S and arranging them in *increasing* order. Thus for $n = 6$ and $r = 3$, one 3-combination is (2 4 6). Two r-combinations are *distinct* if they differ with respect to the items they contain. The number of distinct m-combinations of n items is denoted by $C(n,m)$, where $C(n,m) = \frac{n!}{(n-m)!m!}$.

Thus for $n = 4$, there are four distinct 3-combinations. In the special case where $r = n$, $C(n,n) = 1$.

The 3-combinations of $\{1,2,3,4\}$ in lexicographic order are:

(1 2 3), (1 2 4), (1 3 4), (2 3 4).

Since there are $C(n,r)$ r-combinations, each of length r, combination generation has a trivial lower bound of $\Omega(C(n,r)r)$.

The current best parallel algorithm for the generation of r-combinations in lexicographic order runs with a time complexity of $O(r)$ and is cost optimal[34].

We present three optimal algorithms for the lexicographic generation of r-combinations of n distinct objects in Chapter 7. The first algorithm runs with a time complexity of $O(r)$ and optimal work $O(C(n,r)r)$ on the CREW PRAM and is based on building a *combination tree*. The second algorithm has a time complexity of $O(\log r(\log r + \log\log * n))$ on CREW PRAM and $O(\log r(\log r + \alpha(n))$ on COMMON CRCW PRAM and is based on divide and conquer on the parameter r. The third algorithm runs with a time of $O(\log n)$ on CREW PRAM and is based on divide and conquer on the parameter n.

Derangements

A derangement of S is defined as a permutation P of these integers which changes every element so that no integer appears in its natural position. Formally, if P is the ordered n-tuple $(p_1 p_2 \ldots p_n)$ then P is a derangement of $\{1, 2, \ldots, n\}$ provided that $p_i \neq i$ for all i, $1 \leq i \leq n$. The number of derangements of n objects is denoted by $D(n)$. $D(n)$ satisfies the recurrence relation

$$D(n) = (n-1)(D(n-1) + D(n-2)) \tag{1.1}$$

with $D(0) = 1$ and $D(1) = 0$.

For example, there are nine derangements for $n = 4$, namely

(2 1 4 3), (2 3 4 1), (2 4 1 3), (3 1 4 2),
(3 4 1 2), (3 4 2 1), (4 1 2 3), (4 3 1 2),
(4 3 2 1)

Since there are $D(n)$ derangements, each of length n, derangement generation has a trivial lower bound of $\Omega(D(n)n)$.

The current best parallel algorithm[23] for derangements runs with a time complexity of $O(n \log n)$ with $O(nD(n) \log n)$ work. Obviously this algorithm is not cost optimal.

We present new, efficient and optimal parallel algorithms for the generation of derangements in Chapter 6. The first algorithm runs in $O(n)$ time with optimal work $O(D(n)n)$ on CREW PRAM. This algorithm is based on the concept of *derangement tree*. We present a novel method to show that given an $O(T)$ time and $O(W)$ work algorithm for lexicographic generation of all permutations of n objects, we can automatically generate all derangements of n objects in $O(T + \log n)$ time with $O(W)$ work

on the same model on which permutation generation algorithm runs. This results in an $O(\log n)$ time $O(nD(n))$ optimal work algorithm for lexicographic generation of derangements on EREW PRAM. These improvements are a result of a ranking function, we designed for derangement. The ranking function is also of independent interest. We discuss parallelization of ranking function on EREW and CREW PRAMs. We can rank a derangement in $O(\log n)$ time with $O(n^3)$ work on EREW PRAM. We can rank a derangement in $O(\log n)$ time with $O(n \log n)$ work on CREW PRAM.

1.2 Parallel Computation Models

We will be using the Parallel Random Access Machine(PRAM) model[17]. In this model, processors are numbered $1, 2 \ldots, p$ and various cells in the common shared memory are numbered $1, 2, \ldots$. Each processor knows its own number. In PRAM basic model itself there are different models depending on the reading and writing of memory access strategies. They are given below:

1. **Exclusive-Read, Exclusive-Write(EREW):** Access to memory locations is exclusive. In other words, no two processors are allowed simultaneously to read or write into the same memory location.

2. **Concurrent-Read, Exclusive-Write(CREW):** Multiple processors are allowed to read the same memory location simultaneously. But more than one processor writing into the same memory location simultaneously is forbidden.

3. **Concurrent-Read, Concurrent-Write(CRCW):** More than one processor are allowed to read and write into the same memory location simultaneously. Concurrent write is not immediately meaningful. A sub-classification with respect to the conflict resolution strategy based on the values written is standard.

   (a) **COMMON CRCW:** All the processors simultaneously writing into the same location must write the same value.

   (b) **ARBITRARY CRCW:** If more than one processor are writing into the same memory location simultaneously, then one among them succeeds but no guarantee is given as to the identity of the successful processor.

(c) **PRIORITY CRCW:** The smallest indexed processor succeeds when multiple processors are simultaneously writing into the same location.

A parallel algorithm running with P processors for a time T is said to do a work (or has a *cost*) of TP operations. If this work is within a constant multiple of the lower bound for the sequential algorithm, then the parallel algorithm is said to be *optimally efficient* or *optimal*[29, 25, 18]. A primary goal in parallel computation is to design optimally efficient algorithms that run as fast as possible. Our algorithms in this report are optimal according to this criterion.

We will be frequently using the concept of self simulation of PRAM in the design of our algorithms. A PRAM model is said to be *self-simulating* if an algorithm which takes a time of t_i units with p_i processors can also be implemented on that model in $O(r_i t_i)$ time with $\frac{p_i}{r_i}$ processors. The PRAM models we use in this thesis are known to be self-simulating.

1.3 Preliminaries

We express the time complexity of our algorithms in terms of the order notation defined below:

Definition: $f(n) = O(g(n))$ if and only if there exists positive integer constants c and n_0 such that $f(n) \leq cg(n)$ for all $n \geq n_0$.

Definition: $f(n) = \Omega(g(n))$ if and only if there exists positive integer constants c and n_0 such that $f(n) \geq cg(n)$ for all $n \geq n_0$.

Definition: $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $g(n) = O(f(n))$.

Definition: $f(n) = \omega(g(n))$ if and only if for all positive $c \geq 0$, there exists positive integer constant n_0 such that $f(n) > cg(n)$ for all $n \geq n_0$

We define below two other mathematical functions which are useful in our algorithms. Consider

$$\log^{(i)} n = \begin{cases} \log(\log^{(i-1)} n) & \text{if } i \geq 1. \\ n & \text{if } i = 0. \end{cases}$$

Then $\log^* n$ is defined as shown below.

$$\log^* n = min\{i | \log^{(i)} n \leq 1\} \tag{1.2}$$

Note that,

$$\log^*(n^r) = O(\log^* n + \log^* r) \tag{1.3}$$

if $r \leq n$.

Similarly, Ackermann's function is defined as shown below.

$$\alpha(i,j) = \begin{cases} 2^j & \text{if } i = 1 \\ \alpha(i-1,2) & \text{if } j = 1 \\ \alpha(i-1,\alpha(i,j-1)) & \text{if } i,j \geq 2 \end{cases}$$

The inverse Ackermann's function is defined as shown below

$$\alpha(n) = min\{j | \alpha(j,j) \geq n\} \tag{1.4}$$

Note that

$$\alpha(n^r) = O(\alpha(n) + \alpha(r)) \tag{1.5}$$

if $r \leq n$.

Both the functions, Inverse Ackermann and $\log^* n$, are extremely slow growing functions.

1.4 Organization of Thesis

The rest of the thesis is organized as follows. In Chapter 2, we list and discuss some basic parallel algorithms and techniques that are used in later chapters. We discuss optimal algorithms to generate permutations in Chapter 3. Chapter 4 presents efficient parallel algorithms for generation of permutations on a weaker model (EREW). We discuss an approach to make a given non-optimal algorithm optimal for permutation generation in Chapter 5. This approach works for a particular class of permutation generation algorithms. Fast parallel algorithms for generating derangements appear in Chapter 6. We discuss optimal parallel algorithms for generating combinations in Chapter 7. We end the report with concluding remarks which include some suggestions for further work.

Chapter 2

Basics

In this chapter we will look at the basic techniques and problems that are frequently used in later chapters. We will define the problems and the results associated with the algorithms for them.

2.1 Standard Techniques

In this section we will review parallel algorithmic techniques and certain basic problems that frequently arise in later chapters (and in many other parallel algorithms).

Prefix Computation

Consider the sequence of n elements $\{x_1, x_2, \ldots, x_n\}$ from a set S on which a binary *associative* operator '$\otimes$' is defined. The problem of *prefix computation* is defined as the finding of $s_i = x_1 \otimes x_2 \otimes \ldots x_i, 1 \leq i \leq n$

This problem is solved by organizing the computation in the form of a *balanced binary tree*. Prefix computation problem can be solved in $O(\log n)$ time with linear work $O(n)$ on EREW PRAM[25].

Without doubt, prefix computation is the most frequently used subroutine in parallel algorithms. Consequently, there are a number of applications of it. We use the following variations of prefix computation in the later chapters.

- *Prefix sum*: By considering the binary associative operator to be the usual summation operator $(+)$, we get the prefix sum. Note that, the sum of n elements is nothing but the value of s_n. A problem related to prefix sums is *compaction*.

Out of the given n elements certain elements are marked. The problem is to bring together these marked elements, in the order they are existing in the original sequence, such that they occupy adjacent locations. This problem can be easily solved using prefix sums by assigning 0 to each unmarked element and 1 to each marked element and then applying prefix sum on it. At the end of it, each marked element will know its index in the set of marked elements.

- *Prefix Multiplication*: By considering the binary associative operator to be the usual multiplication operator ($*$), we get the prefix multiplication. We use this operation to find values of factorials.

- *Finding OR*: If we consider the binary associative operator to be boolean OR and the set S as the boolean values $\{0, 1\}$, then the OR of all the n boolean values is given by s_n.

Generalized Prefix Computation

Generalized Prefix Computation [39] is defined as:

Let $f[m]$ and $y[m]$ be given sequences of elements for integers $1 \leq m \leq n$. There is an arbitrary binary associative operator '$\otimes$' on the f-elements; the y-elements can be compared by a linear order '$<$'. The problem is to compute the sequence of general prefixes:

$E[m] = f[j_1] \otimes f[j_2] \ldots \otimes f[j_k]$ where $j_1 < \ldots < j_k$ and $\{j_1, \ldots, j_k\}$ is the set of indices $j < m$ for which $y[j]$ '$<$' $y[m]$, where m is each index from 1 to n.

Generalized prefix computation can be done in $O(\log n)$ time with n processors on CREW PRAM[39].

List Ranking

This is another frequently used subroutine in parallel algorithms comparable to prefix computation. Given a linear linked list having n nodes, the problem of *list ranking* is to find for each node, the number of nodes from that node to the end of the list.

The basic technique used for solving list ranking is *pointer jumping*. This problem can be solved in $O(\log n)$ time with linear work on EREW PRAM[6].

A problem related to list ranking is, we are given a set of linked lists with n nodes, and we want to find for each node, the identity of the *last* node in its list. This problem can be solved in the same bounds as given above for list ranking[25].

Cross Ranking

Given A and B are arrays of length s and p respectively and their elements are in non-decreasing order, the problem of *Cross-ranking(A, B)* is defined as follows. It consists of computing two arrays $X = Rank(A)$ and $Y = Rank(B)$ (the elements of X and Y will be denoted $x_1, x_2, \ldots x_s$ and $y_1, y_2, \ldots, y_p$, respectively) such that

- x_i is the rank of a_i in B, defined as the number of elements in B that are *less* than a_i.
- y_j is the rank of b_j in A defined as the number of elements in A that are *less than or equal to* b_j.

Merging is a related problem in which given two sorted arrays we would like to obtain a sorted array containing all the elements in its input sorted arrays. The problem of cross ranking and merging are equivalent with respect to time and processor bounds[8].

Merging of two lists of combined length n containing integers in the range $[1..n]$ can be done in $O(\log\log * n)$ time with $O(n)$ work on CREW PRAM[8]. The same problem can be solved in $O(\alpha(n))$ time with $O(n)$ work on COMMON CRCW PRAM[8].

Sorting

Given a sequence of n elements $\{x_1, x_2, \ldots, x_n\}$ from a set S on which a linear order is defined, the problem of *sorting* is to find a permutation π such that $1 \leq i \leq j \leq n$ if and only if $x_{\pi(i)} \leq x_{\pi(j)}$.

If *comparison* is the only allowed operation on the elements, then sorting of n elements can be done in $O(\log n)$ time with n processors on EREW PRAM[12]. On the other hand, if the elements are from the domain of integers, we can do better. In particular, n integers drawn from a set $\{0, 1, 2, \ldots, m-1\}$ can be sorted on an ARBITRARY CRCW PRAM [9] in time $O(\log n / \log\log n + \log\log m)$ with work of $O(n \log\log m)$ and in $O(\log n)$ time with $O(n \log\log n)$ work.

2.2 Standard Procedures

In this section, we will look at some of the subroutines that are frequently used in later chapters. We will look at the problems and the algorithms for them in detail.

The operation *ranking* occurs quite commonly in parallel algorithms for combinatorial generation. We present the ranking functions for permutations and combinations followed by their parallelization.

2.2.1 Ranking of Permutation

There exists a one to one correspondence between the integers $1, \ldots, P(n,m)$ and the set of m-permutations of $\{1, 2 \ldots, n\}$ listed in lexicographic order.

Specifically, a function *rankp* with the following properties can be defined [3]:

1. Let $(p_1 p_2 \ldots p_m)$ be one of the $P(n,m)$ permutations of $\{1, 2 \ldots, n\}$. Then $rankp(p_1, p_2 \ldots p_m)$ is an integer in $\{1, 2, \ldots, P(n,m)\}$.

2. Let $(p_1 p_2 \ldots p_m)$ and $(q_1 q_2 \ldots q_m)$ be two m-permutations of $\{1, 2 \ldots, n\}$ then $(p_1 p_2 \ldots p_m)$ precedes $(q_1 q_2 \ldots q_m)$ lexicographically if and only if $rankp(p_1, p_2, \ldots, p_m) < rankp(q_1, q_2, \ldots, q_m)$.

For the permutation $(p_1 p_2 \ldots p_m)$ define the sequence $\{r_1, r_2 \ldots, r_m\}$ as follows: $r_i = p_i - i + \sum_{j=1}^{i-1} [p_i < p_j]$ here $[p_i < p_j] = 1$ if $p_i < p_j$; and $[p_i < p_j] = 0$ if $p_i \geq p_j$. The string $r_1 r_2 \ldots r_m$ can be interpreted as a mixed radix integer where

$0 \leq r_m \leq n - m$

$0 \leq r_{m-1} \leq n - m + 1$

$\vdots$

$0 \leq r_2 \leq n - 2$

$0 \leq r_1 \leq n - 1$

Expressing $r_1 r_2 \ldots r_m$ as a decimal number gives us the integer corresponding to $(p_1 p_2 \ldots p_m)$:

$$rankp(p_1, p_2, \ldots, p_m) = 1 + \sum_{i=1}^{m-1} r_i \prod_{j=0}^{m-i-1} (n - i - j)$$

In the above equation there are m terms where the first term is 1 and the remaining terms are of the form r_i multiplied by a factor for each i from 1 to $m-1$. We can easily see that in the above equation, the product $\prod_{j=0}^{m-i-1}(n-i-j)$ for each value of i can be computed as follows. We use $m-1$ extra locations. Location i should contain the value $n-i$. Find the prefix products on this array in time $O(\log m)$ with $O(\frac{m}{\log m})$ processors [19]. The value in the array location j corresponds to the multiplicand corresponding to r_{m-j} in the above equation. Next consider the computation of values of $r_1 r_2 \ldots r_m$ for an arbitrary permutation $(p_1 p_2 \ldots p_m)$. To calculate r_is the non-trivial term is $\sum_{j=1}^{i-1}[p_i < p_j]$ for each $i, 1 \leq i \leq n$, which in turn is a part of calculations of the number of *inversions* [27] of the given permutation. Calculations of the inversions can be done using the algorithm for *Generalized Prefix Computation* (see 2.1) which takes $O(\log m)$ time with $O(m)$ processors on the CREW model. Once these expressions are evaluated r_is can be calculated in constant time with $O(m)$ processors. Then we can calculate the rank of the permutation (using above equation) in $O(\log m)$ time with $O(\frac{m}{\log m})$ processors. We are using only simultaneous read capabilities. Hence the algorithm can be implemented on the CREW model. We summarize our discussion in the following theorem:

Theorem 2.1 *The rank of an r-permutation of n objects can be obtained in $O(\log r)$ time with r processors on CREW PRAM.*

Note that, if we use only EREW PRAM, the following theorem is not difficult to prove.

Theorem 2.2 *The rank of an r-permutation of n objects can be obtained in $O(\log r)$ time with $O(r^2)$ work on EREW PRAM.*

2.2.2 Ranking of Combination

There exist a one-to-one correspondence between the integers $1, \ldots, C(n,r)$ and the set of r-combinations of $\{1, 2 \ldots, n\}$ listed in lexicographic order. Let $(c_1 c_2 \ldots c_r)$ represent one such combination (where by definition, $c_1 < c_2 < \ldots c_r$). We define[3]

$complement(n, c_1, c_2, \ldots, c_r) = (d_1 d_2 \ldots d_r)$

as the *complement* of $(c_1 c_2 \ldots c_r)$ with respect to $\{1, 2 \ldots n\}$, where

$d_i = (n+1) - c_{r-i+1}$

Now let the *reverse* of $(c_1 c_2 \ldots c_r)$ be given by $(c_r c_{r-1} \ldots c_1)$. The mapping $order(c_1, c_2, \ldots c_r) = \sum_{i=1}^{r} C(c_i - 1, i)$ has the following properties:

1. if $(c_1 c_2 \ldots c_r)$ and $(c_1' c_2' \ldots c_r')$ are two r-combinations of $\{1, 2, \ldots, n\}$ and the reverse of $(c_1 c_2 \ldots c_r)$ precedes the reverse of $(c_1' c_2' \ldots c_r')$ in lexicographic order, then

   $order(c_1, c_2, \ldots c_r) < order(c_1', c_2', \ldots, c_r')$;

2. $order(1, 2, \ldots, r) = 0$ and $order((n-r+1), (n-r+2), \ldots, n) = C(n, r) - 1$ implying that the transformation *order* maps the $C(n, r)$ different r-combinations onto $\{0, 1, \ldots C(n, r) - 1\}$ while preserving reverse lexicographic order.

Using order and complement, we can define the following one-to-one mapping of $C(n, r)$ possible combinations onto $\{1, 2, \ldots, C(n, r)\}$, which preserves lexicographic ordering:

$rankc(n, c_1, c_2, \ldots, c_r) = C(n, r) - order(complement(n, c_1, c_2, \ldots, c_r))$.

Thus $rankc(n, 1, 2, \ldots, r) = 1, rankc(n, 1, 2, \ldots, r+1) = 2, \ldots, rankc(n, (n-r+1), (n-r+2), \ldots, n) = C(n, r)$.

It is easy to note that based on the ranking function, we can find the rank of an r-combination in $O(\log r)$ time using $r/\log r$ processors as we have to sum r values.

2.3 Generic Processor Allocation Problem

In this section, we describe the processor allocation problem which we will be encountering in our algorithms. The problem is formally stated as follows:

Processor Allocation Problem: We have to accomplish s identical tasks but on data sets of different lengths. The number of processors needed by task i are given by $size[i]$ and they are indexed by $start[i]$ to $end[i]$. Given this information, we would like to form two arrays G and N, where $G[i]$ specifies task number for which processor i should work and $N[i]$ specifies the index of the processor i out of all the processors that are allocated

to its task *viz.* $G[i]$. Further, $start[i] = end[i-1] + 1$ for $1 \leq i \leq s$, i.e the tasks use the adjacent set of processors.

Once the information G and N are known, it is easy to allocate the processors[9]. We can also observe that if out of the three arrays *start*, *end* and *size*, any two are known, we can obtain the third one:

$size[i] = end[i] - start[i] + 1$

Let $p = \sum_{i=1}^{i=s} size[i]$, i.e, p is the total number of processors needed for all the tasks. We assume p is available as part of the input to the algorithm.

We proceed as follows. We form an array B such that $B[i] = i$, $1 \leq i \leq p$. The cross ranking of *start* and B provides us with the necessary details. Note that *start* and B are non-decreasing arrays of length s and p respectively and they contain integers in the range $[1..p]$. The problem of cross-ranking(A, B) is defined as follows. It consists of computing two arrays $X = Rank(A)$ and $Y = Rank(B)$ (the elements of X and Y will be denoted $x_1, x_2, \ldots x_s$ and $y_1, y_2, \ldots, y_p$, respectively) such that

- x_i is the rank of a_i in B, defined as the number of elements in B that are less than a_i.
- y_j is the rank of b_j in A defined as the number of elements in A that are *less than or equal to* b_j.

The problem of cross ranking and merging are equivalent[8].

Since $G[i]$ represents the task to which processor i should be allocated, is equal to the number of elements in A which are less than or equal to i. As $i \geq A[y_i]$ and $i < A[y_{i+1}]$, it follows that, processor i should be allocated for task y_i as it is in the range of $[start[y_i]..end[y_i]]$. It is obvious that $N[i] = i - start[G[i]] + 1$, $1 \leq i \leq p$;

Merging of two lists of combined length n containing integers in the range $[1..n]$ can be done in $O(\log\log * n)$ time with $O(n)$ work on CREW PRAM[8]. The same problem can be solved in $O(\alpha(n))$ time with $O(n)$ work on COMMON CRCW PRAM[8].

Note that the only operation that takes non-constant time in the above algorithm is cross ranking.

Thus, we obtain the following theorems.

Theorem 2.3 *The Generic processor allocation problem can be solved with a time complexity of* $O(\log\log*(s+p))$ *and* $O(s+p)$ *work on the CREW PRAM.* ∎

Theorem 2.4 *The Generic processor allocation problem can be solved with a time complexity of* $O(\alpha(s+p))$ *and* $O(s+p)$ *work on the COMMON CRCW PRAM.* ∎

Chapter 3

Faster Optimal Parallel Algorithms for the Generation of Permutations

3.1 Introduction

The problem of generating permutations has a long history, and a large number of sequential algorithms exist for its solution. This history is traced in [38] along with a review of the different approaches. Parallel permutation algorithms have been designed for SIMD computers with no communication between processors[22] and for linear processor arrays[5, 11]. A cost optimal algorithm employing up to $P(n,r)/n$ processors is presented in [2]. Another cost optimal algorithm with $O(n)$ time complexity is presented in [34]. In this chapter, we discuss new parallel algorithms for enumerating permutations.

We begin with some definitions. Let S be a set consisting of n distinct items say, the first n positive integers, i.e, $S = \{1, 2, \ldots, n\}$. An *r-permutation* of S is obtained by selecting r distinct integers out of S and arrange them in some order. Thus, for example, for $n = 10$ and $r = 4$, a 4-permutation might be (5 7 9 3). Two r-permutations are different (or *distinct*) if they differ either with respect to the items they contain or with respect to the order of items. The number of distinct r-permutations of n items is denoted by $P(n,r)$, where $P(n,r) = \frac{n!}{(n-r)!}$. Thus, for $n = 4$, there are twenty four distinct 3-permutations. In the special case, where $r = n$, $P(n,n) = n!$.

Now, let $x = (x_1 x_2 \ldots x_r)$ and $y = (y_1 y_2 \ldots y_r)$ be two r-permutations of S. We say that x *precedes* y *in lexicographic order* if either $x_1 < y_1$ or if there exists an integer

i, $1 \leq i \leq r$, such that $x_j = y_j$ for all $j < i$ and $x_i < y_i$.

Each r-permutation has an associated index in lexicographic order. By *ranking* we mean getting the index of an arbitrary permutation in lexicographic order. Similarly, *unranking* means getting the r-permutation given its index in the lexicographic order. Thus, unranking is the inverse operation of ranking.

Note that lexicographic order is a *total order* [42](i.e, if A and B are two permutations, then either $A \prec B$ or $A = B$ or $A \succ B$; we use the notation '$\prec$' to represent the relation lexicographic precedence).

In Sections 2 and 3 we describe two algorithms which are much faster when compared to the existing algorithms. Both algorithms are designed based on induction and recursion and are inspired by [32] in which induction and recursion are used as a design technique to derive new and efficient algorithms.

For generation of r-permutations of n objects there are two parameters *viz.* r and n. The recursion can be applied on either of these parameters. The algorithm in section 3 correspond to recursion on the parameter r (r-recursive algorithm) while that of Section 4 correspond to the parameter n (n-recursive algorithm).

3.1.1 Preliminaries

In this section, we discuss few basic concepts related to permutations for later easy reference. The reader is referred to [26] for proofs.

$$P(n,r) = \frac{n!}{(n-r)!} \tag{3.1}$$

$$P(n,r) = n(n-1)\ldots(n-r+1) \tag{3.2}$$

$$P(n,r) = 0, \; if \; n < r. \tag{3.3}$$

$$P(n_1,r) \leq P(n_2,r), \; if \; n_1 \leq n_2 \tag{3.4}$$

$$P(n,r_1) \leq P(n,r_2), \; if \; 0 \leq r_1 \leq r_2 \leq n \tag{3.5}$$

We assume that

$$P(n,r) = 0, \ if \ r < 0 \ or \ n < 0 \tag{3.6}$$

$$C(n,r) = \frac{n!}{(n-r)!r!} \tag{3.7}$$

3.2 The r-Recursive Algorithm

3.2.1 Basic Algorithm

Suppose we are interested in generating all the r-permutations of n objects in lexicographic order. The main idea behind the algorithm is we generate r-permutations if $\frac{r}{2}$-permutations of n objects are available to us; thus, the algorithm is recursive. The termination of recursion is when $r = 1$ in which case obtaining the 1-permutations of n objects is trivial; the 1-permutations of n objects are the n objects in order. The algorithm is given below.

```
Pre-Processing
// Calculate P(s,t) for 1 ≤ s ≤ n and 1 ≤ t ≤ r.
Store these values in a two dimensional array perm.
1 for i = 1 to n*r in parallel do
      // group and shift are local variables of each processor
      /* form n arrays of length r each, filled by consecutive
      integers with starting value as i, 1 ≤ i ≤ n */
      group = (i-1) div r + 1;
      shift = (i-1) mod r + 1;
      val[(group -1) * r + shift] = group + shift - 1;
2 Calculate prefix multiplications of n segments of array val
independently and simultaneously, where each segment is of
length r and the i-th segment starts at (i-1)*r + 1
3 for s = 1 to n in parallel do
      for t = 1 to r in parallel do
```

```
          if t > s then perm[s,t] = 0
          else perm[s,t] = val[(s-t)*r + t];
-----------------------------------------------------------------
Algorithm Generate_Permutations(n,r)
Begin
4a)   if r = 1 then
         for i = 1 to n in parallel do
             A[i,1] = i;
4b)   else
          Generate_Permutations(n,r/2);
          Double_Size();
End
Double_Size()
Begin
5     Copy the r/2-permutations from array A to a separate array B;
      // obtain the r/2-permutations of the n - r/2 objects {1,2,...,n - r/2}
6          Mark those r/2-permutations in array B which have an
                item greater than n - r/2;
7          Rank all the unmarked r/2-permutations of array B storing them in array C
8     For each r/2-permutation of array A obtain the sorted list of its items;
9     For each r/2-permutation obtain the sorted list of unused elements;
10    Form r-permutations by matching ;
End
```

The algorithm as given works under the following assumption.

Assumption: r must be a power of 2.

We relax this restriction in Section 3.2.2.

We will initially describe the pre-processing steps. In the pre-processing we find the values of $P(s,t)$ when $1 \leq s \leq n$ and $1 \leq t \leq r$. To complete pre-processing in $O(\log r)$ time, we find the prefix multiplication of range of length r for each starting value from 1 to n. This has been accomplished by an application of standard prefix multiplication algorithm [19] on these different sets independently and simultaneously.

Once these values are found it is easy to find the values of $P(s,t)$ for the given ranges of s and t as already specified using Eq. (3.2). This can be done in constant time.

The main algorithm is recursive in nature. The base case of the recursion is when $r = 1$ is easy; the 1-permutations of n objects are simply the n objects in lexicographic order. *Double_Size()* when given all $\frac{r}{2}$-permutations obtains all r-permutations.

First consider the following example. Consider the 2-permutations of 4 objects $\{1,2,3,4\}$ in lexicographic order (top to bottom, left to right):

(1 2) (2 3) (3 4)
(1 3) (2 4) (4 1)
(1 4) (3 1) (4 2)
(2 1) (3 2) (4 3)

Consider the 4-permutations of 4 objects $\{1,2,3,4\}$ shown below.

(1 2 3 4) (2 3 1 4) (3 4 1 2)
(1 2 4 3) (2 3 4 1) (3 4 2 1)
(1 3 2 4) (2 4 1 3) (4 1 2 3)
(1 3 4 2) (2 4 3 1) (4 1 3 2)
(1 4 2 3) (3 1 2 4) (4 2 1 3)
(1 4 3 2) (3 1 4 2) (4 2 3 1)
(2 1 3 4) (3 2 1 4) (4 3 1 2)
(2 1 4 3) (3 2 4 1) (4 3 2 1)

Let us consider r-permutations of n objects in lexicographic order. Considering their prefixes of length $\frac{r}{2}$ in order, we observe that:

Lemma 3.1 *For r-permutations in lexicographic order the prefixes of length i (when $i \leq r$) and adjacent duplicates are removed, are in lexicographic order and they are the i-permutations of n objects $\{1,2,\ldots,n\}$.*

Proof: Lemma can be easily proven by contradiction. ∎

From Lemma 3.1, for each $\frac{r}{2}$-permutation there will be a set of r-permutations with this $\frac{r}{2}$-permutation as prefix. Further, all these r-permutations will be together.

Consider the prefixes of length i of an r-permutation of n objects. Let it be $c_1, c_2, \ldots, c_i$. The prefix is an i-permutation of n objects. Thus, every prefix of length i of any r-permutation is an i-permutation of n objects.

In the above example, with 1 2 as a prefix there are two 4-permutations. In fact, with a given prefix of length i, there will be $P(n-i, r-i)$ r-permutations. We consider the $\frac{r}{2}$-permutations as prefixes of the potential r-permutations, and use them to obtain all r-permutations, i.e we find the suffixes that should be added to each $\frac{r}{2}$-permutation to make them r-permutations.

Theorem 3.2 *Given p-permutations of n objects in lexicographic order and given any arbitrary q-permutation $Q = (c_1 c_2 \ldots c_q)$ of n objects. There will be $P(n-q, p-q)$ p-permutations with this q-permutation as prefix. The suffixes that are appended to Q to form the p-permutations are the $(p-q)$-permutations of $n-q$ objects $\{1, 2, \ldots, n\} - \{c_1, c_2, \ldots, c_q\}$. More over these p-permutations are lexicographically consecutive in the given p-permutations, i.e, they are ordered such that the suffixes of length $p-q$ are ordered lexicographically.*

Proof: In the theorem, by lexicographically consecutive, we mean those p-permutations which have consecutive ranks in lexicographic order; the p-permutations with the same prefix are together in lexicographic order.

We first prove the *existence* of $P(n-q, p-q)$ p-permutations with the given prefix. Then, we prove that they are adjacent lexicographically,

Let us consider the given q-permutation of n objects $Q = (c_1 c_2 \ldots c_q)$. Let $s = p-q$. Let us also consider the s-permutations of the $n-q$ objects $B = \{1, 2, \ldots, n\} - \{c_1, c_2, \ldots, c_q\}$. Let $a = (a_1 a_2 \ldots a_s)$ be an arbitrary s-permutation of B. Append the s-permutation a to Q to form D. D is a p-permutation of n objects as all the items in D are distinct and are coming from the set of n objects $\{1, 2, \ldots, n\}$ and is of length p. Since, we have started with an arbitrary s-permutation, it follows that each of the s-permutations of $n-q$ objects can be appended to Q getting a p-permutation of n objects. Further, all such formed p-permutations are distinct.

Now, we prove that whenever a p-permutation is there with Q as prefix, its suffix of length s is a s-permutation of B. Suppose $u = (u_1 u_2 \ldots u_p)$ is a p-permutation with Q as prefix. Obviously, $u_i = c_i$ for $1 \leq i \leq q$. Let us consider the part $(u_{q+1}, \ldots, u_p)$ i.e the suffix of length $s = p-q$ of the above p-permutation. The elements of the suffix must be coming from the set B as in a permutation all the elements must be distinct. From this it follows that the above suffix is a s-permutation of B.

Since there are $P(n-q,p-q)$ s-permutations of B, there will be $P(n-q,p-q)$ p-permutations with the prefix as Q.

From our above claims, we noted that the p-permutations, with the given prefix have the suffixes which are nothing but the s-permutations of $n-q$ objects B. Now, it remains for us to prove that all these p-permutations with our prefix will be lexicographically adjacent if we consider the total p-permutations of n objects.

We prove this in two steps. We first prove that all our p-permutations will be together in the complete p-permutations in lexicographic order. Later we prove that they are adjacent lexicographically (consecutive ranks). We prove this by contradiction. Suppose they are not together. Then this means that there exists three p-permutations B_1, B_2 and B_3 such that B_1, B_3 have the prefix $t_1 = (c_1 c_2 \ldots c_q)$ and B_2 has a different prefix t_2 of length q and B_1, B_2 and B_3 are in lexicographic order (not necessarily adjacent). Since t_1 and t_2 are different and B_1 precedes B_2 lexicographically it follows that t_1 precedes t_2 lexicographically. But this implies B_3 precedes B_2 lexicographically. A contradiction. Hence all the p-permutations with the same prefix are together.

Once we have proved that the p-permutations with the prefix as our original q-permutation are together it is easy to prove by contradiction that they are adjacent lexicographically such that the suffixes of length $p-q$ are in lexicographic order.

Note that our complete p-permutations are in lexicographic order. In that p-permutations with prefix as Q are together and are in lexicographic order. We have to prove that the suffixes of our p-permutations with the given prefix are ordered lexicographically. Suppose they are not. Then there exist two p-permutations w and x such that both have the same prefix $(c_1, c_2, \ldots, c_q)$ and have suffixes such that they are not in lexicographic order while w and x are in order lexicographically. Since both have the same prefix if the suffixes are not in lexicographic order then by definition, w and x can not be in lexicographic order. Hence a contradiction.

Putting all this together our claims follow. ∎

As a corollary to the above theorem, we have

Corollary 3.3 *Let $(a_1 a_2 \ldots a_{\frac{r}{2}})$ be $\frac{r}{2}$-permutation of n objects. The suffixes that should be added to this $\frac{r}{2}$-permutation to make the corresponding r-permutations are the $\frac{r}{2}$-permutations of $n-\frac{r}{2}$ objects $\{1,2,\ldots,n\} - \{a_1, a_2, \ldots, a_{\frac{r}{2}}\}$.* ∎

So, once we know the suffixes that should be added to each $\frac{r}{2}$-permutation then it is easy to obtain the r-permutations of n objects. We next address the question of getting the above $\frac{r}{2}$-permutations of $n - \frac{r}{2}$ objects $\{1, 2, \ldots, n\} - \{a_1, a_2, \ldots, a_{\frac{r}{2}}\}$. Note that for each $\frac{r}{2}$-permutation P, the suffixes are coming from the $\frac{r}{2}$-permutations of a set of $n - \frac{r}{2}$ objects which is a subset of $\{1, 2, \ldots, n\}$ depending on P. We have with us the $\frac{r}{2}$-permutations of n objects $\{1, 2, \ldots, n\}$. The following lemma is required.

Lemma 3.4 *If we have the p-permutations of u objects $\{1, 2, \ldots, u\}$ in lexicographic order, then the p-permutations of any other set $\{e_1, e_2, \ldots, e_u\}$ can be obtained as follows:*

1. *Sort the elements of the set $\{e_1, e_2, \ldots, e_u\}$ to get the set $\{g_1, g_2, \ldots, g_u\}$.*
2. *Map the element g_i to i, and in p-permutations of u objects $\{1, 2, \ldots, u\}$ replace i with g_i.*

Then the resulting permutations are the p-permutations of u objects $\{e_1, e_2, \ldots, e_u\}$ in lexicographic order.

Proof: The proof is based on the fact that $i < j$ (here i and j are elements of the set $\{1, 2, \ldots, u\}$) if and only if $g_i < g_j$.

If a is a p-permutation of $\{1, 2, \ldots, u\}$ then we obtain a permutation a' by replacing i with g_i in the above p-permutation. The p-permutation we have got is a p-permutation of $\{e1, e2, \ldots, eu\}$.We denote by m' the corresponding permutation of $\{e1, e2, \ldots, eu\}$ where m is the permutation of $\{1, 2, \ldots, u\}$.

From this fact and definition of lexicographic order (see Section 3.1), we can easily prove that if a p-permutation w precedes another x of $\{1, 2, \ldots, u\}$ then their corresponding permutation w' precedes x' in lexicographic order. ∎

Thus, if we have with us the $\frac{r}{2}$-permutations of the set of $n - \frac{r}{2}$ objects $\{1, 2, \ldots, n - \frac{r}{2}\}$ then we can get easily obtain the $\frac{r}{2}$-permutations of any other set with the same cardinality. So, we next look at the problem of getting $\frac{r}{2}$-permutations of the set of $n - \frac{r}{2}$ objects $\{1, 2, \ldots, n - \frac{r}{2}\}$.

Note that we have with us the $\frac{r}{2}$-permutations of n objects $\{1, 2, \ldots, n\}$. We can see that every $\frac{r}{2}$-permutation of $n - \frac{r}{2}$ objects $\{1, 2, \ldots, n - \frac{r}{2}\}$ must be there in $\frac{r}{2}$-permutations of n objects. More over, it will be occurring only once as part of $\frac{r}{2}$-permutations. But they will be scattered throughout the $\frac{r}{2}$-permutations of n objects. So, we have to bring together these $\frac{r}{2}$-permutations of $\{1, 2, \ldots, n - \frac{r}{2}\}$. This we will do as follows.

We consider the elements $n - \frac{r}{2} + 1, \ldots, n$ as *dummy* elements in the above $\frac{r}{2}$-permutations. So for each $\frac{r}{2}$-permutation, we will *mark* it if it contains at least one dummy element. This can be done by assigning $P(n, \frac{r}{2}).\frac{r}{2}$ processors such that each item gets one processor and that processor decides whether its item is dummy or not. Later using logical OR all the processors for an $\frac{r}{2}$-permutation can decide whether that $\frac{r}{2}$-permutation should be marked or not.

Once we marked the unwanted $\frac{r}{2}$-permutations, we have to *rank* all the unmarked $\frac{r}{2}$-permutations to form the $\frac{r}{2}$-permutations of $n - \frac{r}{2}$ objects $\{1, 2, \ldots, n - \frac{r}{2}\}$. It is important to realize that even though we should rank only the unmarked $\frac{r}{2}$-permutations, processors are assigned to marked $\frac{r}{2}$-permutations nevertheless; these processors remain idle. This is done to avoid the processor allocation complication that may arise if we want to assign processors only to the unmarked permutations. Now we look at the problem of getting *rank* of a permutation to locate its index in lexicographic order.

We can calculate the rank of the permutation in $O(\log m)$ time with $O(\frac{m}{\log m})$ processors (see Section 2.2.1) on CREW PRAM..

Using this ranking procedure, we rank all the unmarked $\frac{r}{2}$-permutations. Then, we store the unmarked $\frac{r}{2}$-permutations in an array C at the position indicated by their rank. So, array C contains the $\frac{r}{2}$-permutations of $n - \frac{r}{2}$ objects $\{1, 2, \ldots, n - \frac{r}{2}\}$ in lexicographic order.

After obtaining the $\frac{r}{2}$-permutations of $n - \frac{r}{2}$ objects $\{1, 2, \ldots, n - \frac{r}{2}\}$ we use Corollary 3.3 and Lemma 3.4 in conjunction for the generation of r-permutations of n objects; basically, for each $\frac{r}{2}$-permutation we would like to attach the corresponding suffixes as defined by Corollary 3.3 to obtain the r-permutations.

Consider an arbitrary $\frac{r}{2}$-permutation $P = (a_1 a_2 \ldots a_{\frac{r}{2}})$ of n objects $\{1, 2, \ldots, n\}$. We call the set $\{a_1, a_2, \ldots, a_{\frac{r}{2}}\}$ the set of *used elements*. Similarly, $M_p = \{1, 2, \ldots, n\} -$

$\{a_1, a_2, \ldots, a_{\frac{r}{2}}\}$ is called the set of *unused elements*. We would like to determine for each $\frac{r}{2}$-permutation its *sorted* list of unused elements.

We discuss the finding of this for an arbitrary $\frac{r}{2}$-permutation. The procedure can then be applied independently and simultaneously to each $\frac{r}{2}$-permutation.

Let $(a_1 a_2 \ldots a_{\frac{r}{2}})$ be an arbitrary $\frac{r}{2}$-permutation. We sort these elements using Cole sort [12] to get the sorted list of elements say, $B = (b_1, b_2, \ldots, b_{\frac{r}{2}})$. Let B_p be an array of length n initialized with zeroes. We mark $B_p[a_i]$ for all i in this array. This can be done in constant time using $\frac{r}{2}$ processors. Thus, the unused elements are the unmarked elements in the array B_p. Let A_p be another array of length n with $A_p[i] = i$, for all $1 \leq i \leq n$. From $Cross - Rank(B, A_p)$ (see Section 2.1), we get an array Y such that y_i is the number of elements in B which are less than or equal to $A_p[i] = i$. For each unused element we intend to find its position in the sorted list. Now, consider an arbitrary i. If $B_p[i] = 1$, then we can ignore i as it is an used element. On the other hand, if $B_p[i] = 0$, then proceed as follows. Note that i does not belong to B. There are y_i used elements that are less than i; there are $i - y_i$ unused elements that are less than or equal to i. Hence, the index of i in the list of unused elements is $i - y_i$. Let $C_p[i - y_i] = i$.

We use Lemma 3.4 to form the r-permutations. This is done in two steps.

In the first step we assign $P(n - \frac{r}{2}, \frac{r}{2}).\frac{r}{2}$ processors to each $\frac{r}{2}$-permutation and for all i, $1 \leq i \leq P(n, \frac{r}{2})$, copy the i^{th} $\frac{r}{2}$-permutation $P(n - \frac{r}{2}, \frac{r}{2})$ times and place these in locations $(i-1)P(n - \frac{r}{2}, \frac{r}{2}) + 1$ to $iP(n - \frac{r}{2}, \frac{r}{2})$ of the array E. Here E is a two dimensional array of length r in its second dimension. The permutations occupy locations 1 to $\frac{r}{2}$ in its second dimension.

In the second step, we again assign $P(n - \frac{r}{2}, \frac{r}{2}).\frac{r}{2}$ processors to each $\frac{r}{2}$-permutation. Each processor copies the $\frac{r}{2}$-permutations of $n - \frac{r}{2}$ objects $\{1, 2, \ldots, n - \frac{r}{2}\}$ with *proper matching*. By proper matching we mean suppose a processor is copying an element i then it will put $C_p[i]$ corresponding to it where this processor is devoted to p th $\frac{r}{2}$-permutation. More over these $P(n - \frac{r}{2}, \frac{r}{2}).\frac{r}{2}$ processors copy into locations $(i-1)P(n - \frac{r}{2}, \frac{r}{2}) + 1$ to $iP(n - \frac{r}{2}, \frac{r}{2})$ of array E except that the locations are $\frac{r}{2} + 1$ to r in the second dimension of it. Once this processing is finished, $E[i]$ is the i th r-permutation of the n objects $\{1, 2, \ldots, n\}$ in lexicographic order for all i, $1 \leq i \leq P(n, r)$.

We next analyze the algorithm. First, consider the pre-processing part of the algorithm. In the first step we are initializing tables which needs constant time and nr processors. In the next step we are finding the prefix multiplications of a number of segments of equal length simultaneously and independently. We are using the standard algorithm for prefix multiplication [25]. It takes $O(\log r)$ time as each segment is of length r. The total work done is $O(nr)$ as the prefix multiplication algorithm is optimal. Then, we find the values of $P(s,t)$ where $1 \leq s \leq n$ and $1 \leq t \leq r$. This is calculated using Eq. (3.2) and can be done in constant time with nr processors by a table look-up.

Thus, the total time for the pre-processing is $O(\log r)$ with $O(nr)$ work.

Let $T(n,r)$ and $W(n,r)$ denote the time complexity and the work done by the algorithm respectively when we are generating r-permutations of n objects.

The algorithm is recursive and the base case of the recursion occurs when we need to generate the 1-permutations of n objects. This we can do in constant time with n processors (see Step 4a). If not, we recursively call the algorithm on n and $\frac{r}{2}$. Obviously this takes a time of $T(n,\frac{r}{2})$ and a work of $W(n,\frac{r}{2})$.

Double_Size obtains r-permutations from $\frac{r}{2}$-permutations. We assume that the $\frac{r}{2}$-permutations are available in an array. In Step 5 we are copying the $\frac{r}{2}$-permutations to a different array B. This can be done in constant time with $P(n,\frac{r}{2}).\frac{r}{2}$ processors. We are marking those $\frac{r}{2}$-permutations which contain a value greater than $n-\frac{r}{2}$ in Step 6. This takes a time of $O(\log r)$ and work of $O(P(n,\frac{r}{2}).\frac{r}{2})$, as marking involves finding the OR of $\frac{r}{2}$ items for each $\frac{r}{2}$-permutation. In Step 7 we are ranking all the unmarked $\frac{r}{2}$-permutations using the procedure described in Section 3.2.1. We are assigning $\frac{r}{2}$ processors to each $\frac{r}{2}$-permutation (not necessarily unmarked permutations) but only that those assigned to unmarked $\frac{r}{2}$-permutations will be finding the rank and others will be idle. Thus, ranking and the consequent storing of all these unmarked permutations in order into array C takes a time of $O(\log\frac{r}{2})$ with $P(n,\frac{r}{2}).\frac{r}{2}$ processors.

In Step 8, we are sorting each $\frac{r}{2}$-permutation independently and simultaneously using Cole's merge sort [12]. This takes $O(\log\frac{r}{2})$ time and $O(P(n,\frac{r}{2})\frac{r}{2}\log\frac{r}{2})$ work. In Step 9, we use merge to get the sorted list of unused elements of each $\frac{r}{2}$-permutation. Thus, this can be implemented in $O(\log\log^* n)$ time with $O(P(n,\frac{r}{2}).n)$ work on CREW PRAM[8].

We are obtaining the final r-permutations of n objects in lexicographic order, by

assigning $P(n-\frac{r}{2},\frac{r}{2}).r$ processors to each $\frac{r}{2}$-permutation in constant time. In other words Step 10 can be implemented in constant time with $P(n,\frac{r}{2}).P(n-\frac{r}{2},\frac{r}{2}).r = P(n,r).r$ processors.

From the above discussion, we form the following recurrence relations.

$$T(n,r) = \begin{cases} T(n,\frac{r}{2}) + c_1 \log r + c_2.\log\log^* n + c_3 & \text{when } r > 1 \\ c_4 & \text{when } r = 1 \end{cases}$$

where c_1, c_2, c_3 and c_4 are constants.

There are $\log r$ levels of recursion. As shown above each level takes $O(\log r + \log\log^* n)$ time. Thus [13],

$T(n,r) = O(\log r(\log r + \log\log *n))$

Similarly for the work performed by the algorithm, the following recurrence relation holds:

$$W(n,r) = \begin{cases} W(n,\frac{r}{2}) + k_1 P(n,\frac{r}{2})\frac{r}{2} + k_3 P(n,\frac{r}{2})\frac{r}{2}\log\frac{r}{2} + +k_5 P(n,\frac{r}{2})n + k_6 P(n,r)r & \text{when } r > 1 \\ k_7 n & \text{when } r = 1 \end{cases}$$

where k_1, k_3, k_5, k_6 and k_7 are constants.

As $r \leq n, \frac{r}{2} \leq \frac{n}{2}$.

Let us consider

$P(n,r)/P(n,\frac{r}{2}) = \frac{n(n-1)(n-2)\ldots(n-r+1)}{n(n-1)(n-2)\ldots(n-\frac{r}{2}+1)} = (n-\frac{r}{2})(n-\frac{r}{2}-1)\ldots(n-r+1)$

There are $\frac{r}{2}$ terms in this expression, and $\frac{r}{2} \geq 1$.

Since $\frac{r}{2} \leq \frac{n}{2}, n-\frac{r}{2} \geq \frac{n}{2}$,

$$\frac{P(n,r)}{P(n,\frac{r}{2})} \geq \frac{n}{2} \tag{3.8}$$

when $r > 1$ and r is a power of 2.

Note that $\log\frac{r}{2} \leq \frac{n}{2}$.

So using Eq. (3.8) and Eq. (3.5) we obtain

$$W(n,r) \leq W(n,\frac{r}{2}) + k.P(n,r).r \tag{3.9}$$

when $r > 1$.

where k is a constant.

We formally show that $W(n,r) \leq 2k.P(n,r).r$

Proof: The proof is based on induction on r.

As base case consider when $r = 1$. From the algorithm, we know that $W(n,1) = k_7.n \leq 2k.n = 2k.P(n,1).1$ when $2k \geq k_7$.

For the induction hypothesis, let us assume that $W(n,r) \leq 2k.P(n,r).r$.

Let us consider the case $2r \leq n$.

So, $W(n,2r) \leq W(n,r) + k.P(n,2r)2r \leq 2k.P(n,r).r + k.P(n,2r).2r$,

using induction hypothesis and Eq. 3.9.

So, $W(n,2r) \leq k.P(n,r).2r + k.P(n,2r).2r \leq k.P(n,2r).2r + k.P(n,2r).2r$

using Eq. (3.5)

$= 2k.P(n,2r).2r$

Hence our claim. ∎

Generation of r-permutation of n objects has a lower bound of $O(P(n,r).r)$. This is the size of output. So, our algorithm is work optimal. Thus, we obtain the following lemma.

Lemma 3.5 *Algorithm works with a time complexity of* $O(\log r(\log r + \log\log * n))$ *with optimal work* $O(P(n,r)r)$ *on CREW PRAM given* r *is a power of* 2.

Note that we can merge two arrays of combined length n can be done in $\alpha(n)$ time when the elements are integers in the range $1..n$ (see Section 2.1). Thus we get the following lemma:

Lemma 3.6 *Algorithm works with a time complexity of* $O(\log r(\log r + \alpha(n)))$ *with optimal work* $O(P(n,r)r)$ *on CREW PRAM given* r *is a power of* 2.

3.2.2 Generalization of the Algorithm

In the above section, we have seen that the r-permutations will be generated only when r is a power of 2. In this section, we generalize that algorithm for the case where r is not a power of 2.

This generalized algorithm is very similar to the original restricted algorithm. In fact, it uses the original algorithm in itself. More over the approach that is used in the

generalized algorithm in its basic form is same as that of the restricted algorithm. The algorithm is given below.

```
1 // Calculate all the values of P(s,t) where for 1 ≤ s ≤ n and 1 ≤ t ≤ r.
Algorithm Generate_Permutations_General(n,r)
Begin
2   r1 is the maximum integer such that
       a) r1 is a power of 2
       b) r1 <= r
       c) Let r2 = r - r1
3   Generate_Permutations(n,r1);
4   Increase_Size(a, r1, r2);
End

Increase_Size(a, r1, r2)
Begin
5   Obtain the r2-permutations of {a+1, a+2, ..., n}; // r1 = a + r2
6   Obtain the r2-permutations of {1, 2, ..., n-a};
7   Obtain the r2-permutations of {1, 2, ..., n-r1} by ranking;
8   For each r1-permutation obtain the sorted list of unused elements;
9   Copy the needed values by matching;
End
```

We are interested in generating r-permutations of n objects, when r is *not* a power of 2. We reduce this problem to our earlier one, where r is a power of 2. We find r_1 such that it is the largest integer which is a power of 2 and is less than or equal to r. So r_1 is the value of most significant bit(MSB) in the binary representation of r. We find r_2 such that $r_1 + r_2 = r$.

Observe that

Observations 2 :

1. $r_1 > r_2$
2. $r_1 \geq \lceil \frac{r}{2} \rceil$

3. $r_2 \leq \lfloor \frac{r}{2} \rfloor$

Once we have obtained r_1, we generate the r_1-permutations of n objects using our earlier algorithm as r_1 is a power of 2. These r_1-permutations are stored in a two dimensional array, A. We obtain the r-permutations of n objects using these r_1-permutations We use the procedure *Increase_Size* to obtain r-permutations.

Observe that each r_1-permutation acts as a prefix for $P(n-r_1, r-r_1) = P(n-r_1, r_2)$ r-permutations (see Theorem 3.2). Suffixes that should be attached to any given r_1-permutation $(a_1 a_2 \ldots a_{r_1})$ are the r_2-permutations of $n-r_1$ objects $S' = \{1, 2, \ldots, n\} - \{a_1, a_2, \ldots, a_{r_1}\}$. Hence, if we obtain r_2-permutations of S', we attach them to get the r-permutations of n objects for a given r_1-permutation; So, if we are able to obtain the r_2-permutations of some set of cardinality $n-r_1$ then we can obtain the r_2-permutations of any other set with the same cardinality using Lemma 3.4. Note that if we attach these suffixes to the r_1-permutations in order we will get the r-permutations of n objects in lexicographic order based on Lemma 3.1 and Theorem 3.2.

We next obtain the r_2-permutations of the set $\{1, 2, \ldots, n - r_1\}$. We would like to get the r_2-permutations of the set $\{1, 2, \ldots, n - r_1\}$ from the r_1-permutations of n objects $\{1, 2, \ldots, n\}$ in lexicographic order (which we already have). Let us consider the prefixes of length $a = r_1 - r_2$ of r_1-permutations; $a > 0$ as $r_1 > r_2$. These prefixes are a-permutations of n objects $\{1, 2, \ldots, n\}$. The first a-permutation in lexicographic order for this set of objects is $(12 \ldots a)$. From Lemma 3.1, we know that the r_1-permutations with this a-permutation will be coming first. Also, from Theorem 3.2, we know that the suffixes added to this a-permutation to form r_1-permutations are $(r_1 - a = r_2)$-permutations of $n - a$ objects $\{1, 2, \ldots, n\} - \{1, 2, \ldots, a\} = \{a+1, a+2, \ldots, n\}$. So, if we take the suffixes of length r_2 of the first $P(n-a, r_2)$ r_1-permutations then we have obtained the r_2-permutations of $n - a$ objects $\{a+1, a+2, \ldots, n\}$. Using Lemma 3.4, we obtain from these the r_2-permutations of $n - a$ objects $\{1, 2, \ldots, n - a\}$. Recall that $a < r_1$ as $r_2 > 0$ (since r is not a power of 2.). Hence $n - a > n - r_1$. So, we consider the elements $n - r_1 + 1, \ldots, n - a$ as dummy elements and then we rank the r_2-permutations which do not contain any dummy element to obtain the r_2-permutations of $n - r_1$ objects $\{1, 2, \ldots, n - r_1\}$.

Once we have obtained the r_2-permutations of $n - r_1$ objects $\{1, 2, \ldots, n - r_1\}$, we

need to get the sorted list of unused elements of each r_1-permutation so that we can attach the suffixes to them to obtain the r-permutations using Lemma 3.4. So, now we want to obtain the sorted list of unused elements for each r_1-permutation.

We proceed in two different ways depending upon the value of r with respect to $\frac{n}{2}$.

Case $r < \frac{n}{2}$

In this case, the way we proceed is similar to the way we have done in our earlier algorithm. For a given r_1-permutation, we first sort its elements and then use cross ranking to get the sorted list of unused elements. (see Section 3.2.1)

Case $r \geq \frac{n}{2}$

Let us consider an arbitrary r_1-permutation $(a_1 a_2 \ldots a_{r_1})$. We use another array I of length n initialized with ones. Then all the r_1 elements of the above r_1-permutation write 0 into their location; i.e, i-th element writes 0 in the location $I[a_i]$. Then apply prefix sums on array I. Suppose j is an unused element then $I[j]$ contains the index of j among the unused elements in sorted order. So, write j into the location $I[j]$ of another array meant for storing these sorted list of unused elements; i.e $C[I[j]] = j$.

If we use the above routine depending on the case for each r_1-permutation, we will obtain the sorted list of unused elements. Once we have got this, we can use Lemma 3.4, to get the r-permutations of n objects.

We will do this in two steps. In the first step for all i, $1 \leq i \leq P(n, r_1)$, we assign $P(n - r_1, r_2).r_1$ processors to i^{th} r_1-permutation and copy that r_1-permutation into locations $(i-1)P(n-r_1, r_2)+1$ to $iP(n-r_1, r_2)$ of the array E. Here E is a two dimensional array of length r in its second dimension. Thus, the permutations occupy the locations 1 to r_1 in its second dimension.

In the second step, we assign $P(n - r_1, r_2).r_2$ processors to each r_1-permutation such that they copy the r_2-permutations of $n - r_1$ objects $\{1, 2, \ldots, n - r_1\}$ with *proper matching.* By proper matching we mean suppose a processor is copying an element j then it will put $C_i[j]$ corresponding to it where this processor is devoted to i^{th} r_1-permutation.

More over these $P(n-r_1,r_2)r_2$ processors copy into locations $(i-1)P(n-r_1,r_2)+1$ to $iP(n-r_1,r_2)$ of array E except that the locations are r_1+1 to r in the second dimension of the array E. Once this processing is finished the array E contains the r-permutations of the n objects $\{1,2,\ldots,n\}$ in lexicographic order. In other words, $E[i]$ contains the i-th r-permutation in lexicographic order.

We analyze the algorithm. Let $T(n,r)$ and $W(n,r)$ denote the time complexity and the work of the generalized algorithm respectively when we are generating r-permutations of n objects.

In Step 1, we are calculating the values of $P(s,t)$ where $1 \leq s \leq n$ and $1 \leq t \leq r$. This will be calculated as in restricted algorithm. Hence it takes $O(\log r)$ time with $O(nr)$ work.

In Step 2, we are finding the value of r_1. It can be obtained in constant time with a single processor. In Step 3, we are making a call to our restricted algorithm with parameters n and r_1. Hence, we obtain the r_1-permutations of n objects in a time of $O(\log r_1(\log r_1 + \log\log *n))$ and a work of $O(P(n,r_1).r_1)$. In Step 4, we are calling *Increase_Size* to obtain the r-permutations from these r_1-permutations.

We are obtaining the r_2-permutations of $\{a+1,a+2,\ldots,n\}$ in constant time with $P(n-a,r_2).r_2$ processors (Step 5). In Step 6, we obtain the r_2-permutations of $\{1,2,\ldots,n-a\}$ in constant time with the same number of processors. In Step 7, we rank, the r_2-permutations to obtain the r_2-permutations of $\{1,2,\ldots,n-r_1\}$. To rank an r_2-permutation, we take a time of $O(\log r_2)$ (see Section 3.2.1) with r_2 processors. In other words, ranking takes a time of $O(\log r_2)$ with $O(P(n-a,r_2).r_2\log r_2)$ work.

In Step 8, we are finding the sorted list of unused elements for each r_1-permutation. Depending upon the value of r with respect to $\frac{n}{2}$ we proceed in two different ways.

If $r < \frac{n}{2}$, then we initially sort each r_1-permutation, which takes a time of $O(\log r_1)$ with $O(P(n,r_1)r_1\log r_1)$ work. Later, we use cross ranking to obtain the sorted list of unused elements which take a time of $O(\log\log *n)$ with $O(P(n,r_1).n)$ work.

On the other hand, if $r \geq \frac{n}{2}$, then we use prefix sums to obtain the sorted list of unused elements which take a time of $O(\log n)$ with $O(P(n,r_1).n)$ work.

In any case, in Step 9, we copy the r_2-permutations to obtain the r-permutations of n objects. This can be done in constant time with $P(n,r).r$ processors.

We consider further analysis as based on the value of r with respect to $\frac{n}{2}$.

Case $r < \frac{n}{2}$

$T(n,r) = O(\log r_1(\log r_1 + \log\log *n)) + c_1.\log r_2 + c_2 + c_3.\log r_1 + c_4.\log\log *n$

where c_1,c_2,c_3 and c_4 are constants.

As $r_1 \leq r$ and $r_2 \leq r$ and using Observation 2, we can easily show that

$T(n,r) = O(\log r(\log r + \log\log *n))$

Regarding the work, we obtain the following expression.

$W(n,r) = kP(n,r_1).r_1 + k_1.P(n-a,r_2).r_2 + k_2.P(n-a,r_2).r_2\log r_2 + k_3.P(n,r_1).r_1\log r_1 + k_4.P(n,r_1).n + P(n,r).r$

where k,k_1,k_2,k_3 and k_4 are constants.

By using Eq. (3.4) and Eq. (3.5) and since $r_1 < r$ and $r_2 < r$ and Observation 2, the above expression can be simplified to

$W(n,r) \leq k.P(n,r).r + k_1.P(n,r).r + k_2.P(n,r_1).r_1\log r_2 + k_3.P(n,r_1).r_1\log r_1 + k_4.P(n,r_1).n + P(n,r).r$

Let us consider

$P(n,r)/P(n,r_1) = \frac{n(n-1)\ldots(n-r+1)}{n(n-1)\ldots(n-r_1+1)} = (n-r_1)\ldots(n-r+1)$

This expression will have $r - r_1$ terms. Since $r < \frac{n}{2}$ and $r > r_1$ it follows that $\frac{P(n,r)}{P(n,r_1)} \geq \frac{n}{2}$.

We know that $\log r_1 \leq \frac{n}{2}$.

Hence, the expression for $W(n,r)$ can be simplified to

$W(n,r) \leq k.P(n,r)r + k_1.P(n,r)r + k_2.P(n,r)r + k_3.P(n,r)r + k_4.P(n,r)r + P(n,r)r$

Hence, $W(n,r) = O(P(n,r).r)$.

Now, we consider the other case,

Case $r \geq \frac{n}{2}$

The expression for time complexity is

$T(n,r) = O(\log r_1(\log r_1 + \log\log *n) + c_1.\log r_2 + c_2 + c_3.\log n$

where c_1, c_2, and c_3 are constants.

Note that since $r \geq \frac{n}{2}$, n is $O(r)$.

As $r_1 \leq r$ and $r_2 \leq r$ and from Observation 2, we can easily show that

$T(n,r) = O(\log r(\log r + \log\log *n))$

Regarding the work, we obtain the following expression.

$W(n,r) = kP(n,r_1)r_1 + k_1.P(n-a,r_2).r_2 + k_2.P(n-a,r_2).r_2 \log r_2 + k_3.P(n,r_1).n + P(n,r).r$ where k,k_1,k_2 and k_3 are constants.

By using Eq. (3.4) and Eq. (3.5) and since $r_1 < r$ and $r_2 < r$ and Observation 2, the above expression can be simplified to

$W(n,r) \leq k.P(n,r).r + k_1.P(n,r).r + k_2.P(n,r_2).r_2 \log r_2 + k_3.P(n,r_1).n + P(n,r).r$

Note that by Observation 2, $r_2 \leq \frac{n}{2}$.

Let us consider

$P(n,r)/P(n,r_2) = \frac{n(n-1)\ldots(n-r+1)}{n(n-1)\ldots(n-r_2+1)} = (n-r_2)\ldots(n-r+1)$

This expression will have $r - r_2$ terms. Since $r > r_2$ and $r_2 \leq \frac{n}{2}$ it follows that $\frac{P(n,r)}{P(n,r_2)} \geq \frac{n}{2}$.

We know that $\log r_2 \leq \frac{n}{2}$.

Hence, the expression for $W(n,r)$ can be simplified to (by using the fact that n is $O(r)$).

$W(n,r) \leq k.P(n,r).r + k_1.P(n,r).r + k_2.P(n,r).r + k_3.P(n,r).r + P(n,r).r$

Hence, $W(n,r) = O(P(n,r).r)$.

By combining the above two cases and Lemma 3.5, we obtain the following theorem

Theorem 3.7 *The r-permutations of n objects can be generated in lexicographic order in time* $O(\log r(\log r + \log\log * n))$ *with optimal work* $O(P(n,r).r)$ *on the CREW PRAM.*

Using Lemma 3.6 we can show that

Theorem 3.8 *The r-permutations of n objects can be generated in lexicographic order in time* $O(\log r(\log r + \alpha(n)))$ *with optimal work* $O(P(n,r).r)$ *on the CREW PRAM.*

3.3 The n-Recursive Algorithm

n this section, we see another algorithm for generating permutations which is also based n induction and recursion.

3.3.1 Basic Algorithm

This algorithm is based on giving an algorithmic interpretation to a well known combinatorial identity. This reinforces our idea that algorithmic interpretation of combinatorial identities provides a good basis through which we can design parallel algorithms.

Our algorithm is based on *Vandermonde's Convolution* for permutations [37],

$$\sum_{k} P(m_1,k)C(r,k)P(m_2,r-k) = P(m_1+m_2,r) \tag{3.10}$$

We use a special case of this identity when $m_1 = \frac{n}{2}$ and $m_2 = \frac{n}{2}$ shown below (assume n is even),

$$\sum_{k=0}^{r} P(\frac{n}{2},k)C(r,k)P(\frac{n}{2},r-k) = P(n,r) \tag{3.11}$$

Our algorithm is based on the following algorithmic interpretation to the above identity.

Suppose, we are interested in generating r-permutations of n objects $\{1,2,\ldots,n\}$. Let $A = (a_1 a_2 \ldots a_r)$ be an arbitrary r-permutation Let there be k objects from the set $\{1,2,\ldots,\frac{n}{2}\}$ in A; $r-k$ objects will be from the set $\{\frac{n}{2}+1,\frac{n}{2}+2,\ldots,n\}$. Let us put the k objects *in the same order* as they are in the permutation A. The places that are occupied by these k elements are a k-combination of the set $\{1,2,\ldots,r\}$. The places occupied by the other $r-k$ elements correspond to the *complement* $r-k$-combination of the above k-combination. In other words one term in the Eq. (3.11) represent one possibility for k. Different values for k lead to different r-permutations. Note that k can have a value from 0 to r only.

Based on this interpretation, we get the following algorithm for generating r-permutations of n objects $\{1,2,\ldots,n\}$. We generate k-permutations for the first set of $\frac{n}{2}$ objects $\{1,2,\ldots,\frac{n}{2}\}$ and $r-k$-permutations for the other set of $\frac{n}{2}$ objects $\{\frac{n}{2}+1,\ldots,n\}$. We also obtain the k-combinations of r objects. Then we put each item of k-permutation in place specified by the k-combination. For each such placement we can place $r-k$-permutations in the remaining places (places specified by the corresponding complement $r-k$-combination). We can generate $P(\frac{n}{2},k).C(r,k).P(\frac{n}{2},r-k)$ r-permutations for any given value of k. Next vary k to generate all the r-permutations of n objects. It is

easy to see that every r-permutation will be definitely generated for some value of k in the range 0 to r.

The algorithm given in this section makes the following assumption.

Assumption: n must be a power of 2.

We will relax this restriction in Section 3.3.2.

By eliminating the zero terms in identity (3.11) using Eq. (3.3) we get

$$\sum_{k=0}^{r} P(\frac{n}{2},k)C(r,k)P(\frac{n}{2},r-k) = P(n,r) \tag{3.12}$$

if $r \leq \frac{n}{2}$.

$$\sum_{k=r-\frac{n}{2}}^{\frac{n}{2}} P(\frac{n}{2},k)C(r,k)P(\frac{n}{2},r-k) = P(n,r) \tag{3.13}$$

if $r > \frac{n}{2}$.

Not surprisingly in the above equations the range of k depends on the value of r.

Each term in the above identity represents two recursive calls. If one of the them produces zero permutations then this means that we have unnecessarily generated both the recursive calls whose result we are not going to use. So we eliminate zero terms from Eq. 3.11 while forming above equations. Note that in the above identity the call to generate i-permutations for j objects occurs only once for applicable i and j.

In identity (3.12) and (3.13) each product is a product of three terms. Two of them represent recursive calls and the other term specifies the combinations that are needed. In our algorithm, we first generate all combinations before proceeding further with the recursive calls. We generate all the i-combinations for j objects $\{1,2,\ldots,j\}$ for all $1 \leq i,j \leq r$ (see Section 3.3.1). Now, we would like to know what are the specific recursive calls that will be made in such an algorithm. It will be useful in the design of our parallel algorithm. In the lemma below, by a recursive call of i-permutations of j objects, we mean any recursive call which occurs such that it generates i-permutations of *some* set of j objects.

Lemma 3.9 *Suppose, we are generating r-permutations of n objects , where n is a power of 2, using Eqs. (3.12), (3.13). As part of the recursive algorithm when we*

generate the p-permutations of m objects then m is power of 2, $m < n$, and $p \leq min(m,r)$.

Proof: The proof is based on the observation that whenever we use Eqs. (3.12) and (3.13), we have eliminated all zero terms in the above identities. More over, in all recursive calls, whenever we are generating q-permutations for $\frac{n}{2}$ objects then $q \leq r$. This applies for every recursive call. Thus, $p \leq r$.

It is given n is a power of 2. It is not difficult to observe that as the recursive calls are for $\frac{n}{2}$ objects, whenever we are generating p-permutations for m objects as part of the complete recursive algorithm, m must also be a power of 2.

If m and p are positive integers $P(m,p) = 0$ only when $m < p$. As there are no zero terms in our identities, it follows that $p \leq m$.

Since $p \leq m$ and $p \leq r$, it follows that $p \leq min(m,r)$. Hence our claim is verified. ∎

As the identities (3.12), (3.13) indicate, we have a number of recursive calls for generation of r-permutations and these recursive calls satisfy the restriction as proved. We have to use parallel recursion, when the number of recursive calls that are to be made are more than one. We avoid parallel recursion by executing the algorithm bottom-up. Thus, we first execute all the calls to obtain the permutations for 1 object. Then we obtain the permutations for 2 objects and so on. We tackle processor allocation details for bottom-up execution in pre-processing. In the pre-processing itself, we will find what processor will do which work for the *entire* main processing. The complete algorithm is shown below.

```
Pre-Processing
// Calculate i! for 0 ≤ i ≤ n
1  for i = 1 to n in parallel do
       A[i] = i;
2  Prefix-mult(A);
   A[0] = 1;
// Calculate values of C(i,j), P(i,j) for all 0 ≤ i,j ≤ n
3  for i = 0 to n in parallel do
```

```
        for j = 0 to n in parallel do
            if i < j then com[i,j] = 0, perm[i,j] = 0
            else com[i,j] = A[i]/(A[i-j]A[j]), perm[i,j] = A[i]/A[i-j];
    // Generating the the i-combinations of j elements {1,2,...,j} for 1 ≤ i,j ≤ r
5   for i = 1 to r in parallel do B[i] = 2^i r;
6   B1 = Prefix-sum(B);
7   for i = 1 to r in parallel do
        start[i] = B1[i-1]+1;
        size[i] = B[i];
        end[i] = start[i] + size[i] -1;
    p = B1[n];
8   Generic-Processor-Allocation;
9   Generate all the i-bit binary numbers for 1 ≤ i ≤ r;
10  Find the combination, complement combination corresponding to each binary
    number using two applications of prefix sum by finding position of 1's,
    and 0's respectively;
11  Rank each combination;
    Put each combination at its index in the list corresponding to its
    combinations.  Take the complementary combination along with it;
//  Pre-processing for Processor allocation of main processing
12  Create an array C of length (log n − 1)r(r + 1).  The locations ((i-1)r(r+1))+1 to
    ir(r+1) correspond to processor allocation details for level i.  If a level i
    starts at position START then the processor allocation for generation of
    m-permutations of level i corresponds to locations START + (m-1)(r + 1) to
    START + m(r+1)-1, where 1 ≤ m ≤ r.
13  D = Prefix-sum(C);
14  for i = 1 to (log n − 1)r(r + 1) in parallel do
        start[i] = D[i-1]+1;
        size[i] = C[i];
        end[i] = start[i] + size[i] -1;
    Total = p = D[(log n − 1)r(r + 1)];
```

```
15 Generic-Processor-Allocation;
-----------------------------------------------------------------
Algorithm Generate_Permutations(n,r)
Begin
16 Generate 1-permutations of 1 object {1} and object {2}
17 for i = 1 to log n − 1 do
       Generate all the permutations related to set of 2^i
       objects and for an adjacent set of elements of the same size.
18 Processor allocation for top level;
19 Generate the r-permutations of n objects;
End.
```

Pre-Processing

In case of main processing we need the values of binomial coefficients and values of $P(i,j)$'s at number of places. We will calculate all the values that are needed and store them.

This we have done in the pre-processing as follows. Apply prefix multiplication[25] on the array A already initialized with $A[i] = i$. After prefix multiplication, $A[i] = i!$. So, once we have calculated the factorial values we use Eq. (3.7) to calculate $C(i,j)$'s and Eq. (3.1) to calculate $P(i,j)$'s for $0 \leq i,j \leq n$.

Generation of Combinations

For generation of combinations using Lemma 3.9 observe:

Observation: If we need i-combinations of j elements in the bottom-up version of recursive algorithm then $i \leq r$ and $j \leq r$.

We generate *all* the i-combinations of j elements for all $0 \leq i,j \leq r$. This means that we have to generate all combinations of r elements and all the combinations of $r-1$ elements and so on. But note that for each combination we also need its complement combination (the *complement* combination of a p-combination of q objects, S is a $q-p$-combination containing all the elements which are in S but not in the p-combination.)

If we are interested in generating all the combinations of k elements, generate all the k-bit binary numbers. Each binary number corresponds to a combination specified by the positions of 1's[33]. The positions of 0's corresponds to the corresponding complement

combination.

As we need to generate combinations for k elements for $k = 1, 2, \ldots, r$, it implies that we have to generate all 2^k k-bit binary numbers where k varies from 1 to r. We generate all binary numbers that are needed at once. Obviously we need k processors to generate a k-bit binary number in constant time. Note that the j-th (counting LSB as 1) bit of a k-bit binary number B is given by $(B \ mod \ 2^j) \ div \ 2^{j-1}$. So, using this expression, we can obtain the binary numbers. But to generate all the k-bit binary numbers we need $2^k.k$ processors for each k from 1 to r. This is not straight forward to do as it is difficult for a processor to know its work. There is a necessity for processor allocation.

To, ease the processor allocation, we allocate r processors instead of k, for each k-bit binary number. The extra processors remain idle. We can imagine the generation of all the k-bit binary numbers for a given k as a task. Hence there are r tasks, such that the i-th task needs $2^i.r$ processors. Based on this idea, we form an array B such that $B[i] = 2^i.r$. We apply prefix sums on this array getting the array $B1$. Now we identify for each task what are the processors that are needed (see Step 7). Then, we apply the generic processor allocation (see Section 2.3), from which we obtain arrays G and N such that $G[j]$ gives the task for which processor j works and $N[j]$ returns the index of the processor j among all the processors that are allocated to task $G[j]$.

Once, processor allocation is done, we will generate all the binary numbers as described above. The task that remains is to obtain the combinations and complement combinations corresponding to each binary number. So, we need to identify the position of each 1 (among all the 1s) and 0 (among all 0s). This can be easily done using two applications of prefix sums.

Note that in generation of permutations we need all the combinations of same length for a given set of elements in sequence. So, our next task is to put together all the p-combinations for j elements for all $0 \leq j, p \leq r$ at one place. This can be done using ranking. As explained in [3] ranking of a k-combination can be done using summation of k values. We are ranking only the combinations but not their complement combinations. We put all the p-combinations of j elements at a place denoted by its rank, we also take its complement combination along with it. When this step is finished, we have got

together all the combinations of the same length for a given set of elements at one place (in consecutive locations).

Main Processor Allocation

We refer the generation of permutations for a particular number of objects as one level. The ith level corresponds to the generation of permutations for 2^i objects. So, there are $\log n + 1$ levels in all from 0 to $\log n$. The level where we are generating the permutations for n objects will be referred to as the *top* level. Similarly, the level where we are generating the permutations for 1 objects will be referred to as the *bottom* level.

Based on Lemma 3.9 we generate all calls such that, for a given level i, we generate p-permutations where p varies from 1 to r. But as we show later whenever $p > 2^i$, no processors will be allocated to this call and in effect the call will not really take place. In accordance with the above idea we will set up processors so that we will execute all the calls as indicated above. All the calls related to a particular level of recursion will execute simultaneously. So, processor allocation should be such that the same processors from one level will be used in the next level.

The processor allocation for a particular call will be based on the Eqs. (3.12) and (3.13). Hence the above equation for the present case provides us with details of how to form the given set of permutations for a given set of objects and also the number of processors needed and how we should distribute them. But we should recall that the above Eqs specifies only the number of permutations and *not* the number of processors that we should devote. But obviously, we will be allocating p processors for each p-permutation. If we multiply by p for the equation to generate p-permutations that will suffice in terms of processor allocation.

Except the top and bottom levels, the remaining levels will be considered in a uniform fashion. For all levels except the top level, we calculate all the p-permutations for p varying from 0 to $min(m, r)$ where we want to generate these p-permutations for m objects.

For each level of recursion, we will try to find the number of processors needed for p-permutations of m objects where p varying from 1 to r. Note that according to Lemma 3.9, p should vary from 1 to $min(m, r)$ and not r. But our modified range for p simplifies the processing with no change in work.

There are $\log n - 1$ levels for which we have to do processor allocation – level 1 to

level $\log n - 1$. For each level, there will be r recursive calls. We create an array C of length $(\log n - 1)r(r+1)$ (recall each recursive call has $r+1$ parts based on Eq. (3.11)) such that the locations $(i-1)r(r+1)+1$ to $ir(r+1)$ correspond to level i. Suppose level i starts at $START$ then the processor allocation details for the jth call in level i correspond to locations $START + (j-1)(r+1)$ to $START + j(r+1) - 1$. These locations will be filled based on Eq. (3.11). Note that even though we are not using Eqs. (3.12), (3.13), the *extra* locations will contain zeroes and hence no processors will be allocated to them any way.

Apply prefix sums on array C. Let the result be in D. We consider each location of D as specifying a task to be done, with the needed number of processors as specified in the array D. Based on this we apply generic processor allocation (see Section 2.3). The calculation of the arrays needed like *start*, *end* and *size* and the value p are specified in the algorithm (see Step 14). Once generic processors allocation is done, we will get the arrays G and N from it.

From now onwards, we refer the array G as *Alloc* and the variable p by $Total$ which denotes the total number of processors that are used in the $(\log n - 1)$ levels.

The idea behind the array *Alloc* is as follows. Let d_i denotes the number of processors we need for level i. More over let e_i 's denote the prefix sums on array d_i. Observe that e_i's can be obtained from the array D. The first d_1 locations of the array *Alloc* provide the processor allocation details for level 1, similarly the next d_2 locations provide the processor allocation details for level 2 and so on. In other words, the locations of *Alloc* array, $e_{i-1}+1$ to e_i provide the processor allocation details for the level i (assume $e_0 = 0$.). For level i, d_i processors will be knowing about work assigned to them by looking at locations $e_{i-1}+1$ to e_i of *Alloc* in order of the processors (1 st processor looks at the location $e_{i-1}+1$, 2nd processor looks at the location $e_{i-1}+2$ and so on.).

We indicate below how a given processor finds the work assigned to it. Suppose a processor numbered j is assigned to the location i of the array *Alloc*. Suppose *Alloc*[i] contains q. Suppose the processor is assigned to the generation of k-permutations. The level in which we are operating is already known as the algorithm proceeds sequentially in terms of levels. Let us also assume that of the k-permutations the processor belongs to t th part of the k-permutations.(Recall that each k-permutation's call has $r+1$ parts;

though some parts may be null, processors will always be assigned to useful parts only). Suppose this t-th part needs some number of processors. So we would like to know what is the number of our processor out of the processors that are allocated for the part t. This is $N[i]$. Once this is known, we can easily identify the work that should be done by our processor.

Main Processing

We generate 1-permutations for 1 object $\{1\}$ and $\{2\}$ to start our processing as they are the base cases in the recursion. The processing for the next first $\log n - 1$ levels is similar. So we describe below the processing for an arbitrary level.

Let us assume we are generating permutations for the i-th level; we will be generating permutations for 2^i elements. There will be two such sets of 2^i elements. More precisely we will be generating permutations for the set $\{1, 2, \ldots, 2^i\}$ and the set $\{2^i + 1, 2^i + 2, \ldots, 2^{i+1}\}$. For the sake of simplicity, let us denote 2^i by w. So, we are generating permutations for first w elements and the next w elements. As we are generating permutations for w elements, we will be generating p-permutations of these w elements (and the other set of w elements) for p varying from 1 to $min(w, r)$. So, there will be $min(w, r)$ calls in all.

From Eq. (3.11), we can observe that, when we want to generate the p-permutations of m elements, we need permutations from the two sets of elements, viz, the first set of $m/2$ elements and the second set of $m/2$ elements; lengths of the permutations that are needed from these two sets are same as Eq. 3.11 is symmetric with respect to its first and second set of elements. The following observation is useful in such situations where we need the same lengths of the permutations for different sets of same cardinality.

Observation: *If we have with us all the p-permutations of the elements $\{1, 2, \ldots, n\}$, then to generate the p-permutations of the set of elements $\{1 + b, 2 + b, \ldots, n + b\}$, we add b to each element of each permutation.*

If we have the p-permutations of one set of elements the p-permutations of the adjacent set with the same size can be easily generated. Coming back to our original discussion, we are interested in generating the p-permutations for w elements and for the adjacent w elements. We do not generate the p-permutations for these two sets independently. One processor will be responsible for one element of a permutation for

the first set of w elements. Now, that processor apart from doing its original duty, adds w to its value (which it is storing in its corresponding location) and stores that value in the corresponding location associated with the p-permutations for the other set of elements.

We now briefly describe how the processors are allocated and are doing their job. For each level, there are corresponding contiguous locations in the array *Alloc*. From the pre-processing we also know, the number of processors needed for each level. For level i, we need $D[ir(r+1)]-D[(i-1)r(r+1)]$ processors. So we allocate that many processors for a given level and the processors identify their work by consulting their assigned location in *Alloc*. Processor j is associated with the location $j' = D[(i-1)r(r+1)]+j$ of *Alloc*. From that they get the needed information. Once the processor knows to which part of p-permutations it is allocated, and its number with respect to the processors that are allocated for a given part i.e $N[j']$, then the processor copies the information from the previous permutations calculated for the last level of processing. Suppose the processor is allocated to part $P(w/2,k)C(p,k)P(w/2,p-k)$; i,e, $P(w/2,k)C(p,k)P(w/2,p-k)p$ processors are needed for this part and these processors copy the k-permutations from the first set of $w/2$ elements placing them in locations identified by the combination and $p-k$-permutations from the second set of $w/2$ elements and placing them in the corresponding positions indicated by the complement combination in a multiplicative fashion.

More precisely, We take each k-permutation from the first set of $w/2$ elements and place it in the positions indicated by k-combinations of p elements $\{1,2,\ldots,p\}$ and for each such placing, we take all the $p-k$-permutations of other set of $w/2$ elements and place them in locations specified by the corresponding complement $p-k$-combination of p elements $\{1,2,\ldots,p\}$. So, there will be $P(w/2,k)C(p,k)P(w/2,p-k)$ p-permutations in all for all the possibilities.

The locations where we have to store these p-permutations will be obtained from the array D. Once a processor knows its number with respect to the part of the p-permutations, it proceeds as follows. It first identifies what is the p-permutation in which it is going to participate. Then it identifies whether it should retrieve an element from the k-permutations of first set of $w/2$ elements or the $p-k$-permutations of the second

set of $w/2$ elements. Suppose it has to copy the j-th element (let it be a') of the k-permutation of first set of $w/2$ elements. It has to note where it should store its element in the p-permutation it is building. This will be identified by the k-combination that is associated with that p-permutation. Then it consults the j-th item of that combination (let it be b') and it stores a' in location b' of the p-permutation.

On the other hand, suppose it has to store the j-th part of the $p-k$-permutation of the other set of $w/2$ elements, then the approach as indicated is applicable except that we use the corresponding complement combination instead of the normal combination to know the location; recall that when we were storing combinations, we store for each combination its complement combination along with it.

In both the above cases, the processor adds w to its element and writes that in the position corresponding to it in the p-permutations for the adjacent set of w elements. The way to identify details like what permutation a processor is associated with, what is the element it should retrieve and where it should store are not difficult to obtain, once we have the information obtained from pre-processing.

For the space allocation, we will have two four dimensional arrays. The first array will be for the first set of elements. The second array will be for the permutations corresponding to the adjacent set of elements. The first dimension of the array correspond to the level of bottom up processing. The second dimension corresponds to the length of permutations, the third dimension correspond to the number of permutation and the fourth dimension corresponds to the actual permutations elements.

Once we have come to top level we have generated all the needed permutations for the first set of $\frac{n}{2}$ elements and the second set of $\frac{n}{2}$ elements. Now we have to calculate the r-permutations of the complete set of n elements from these permutations. We first allocate processors followed by the generation of permutations. We proceed in a similar manner to processor allocation in pre-processing by using the identity (3.12) or the identity (3.13) which ever is applicable. Then we can get r-permutations for n objects.

Now, we analyze the algorithm. Consider the pre-processing part of the algorithm. Initially we are calculating the values of $C(i,j)$'s and $P(i,j)$'s for $0 \leq i,j \leq n$. These are found by finding $i!$ for $1 \leq i \leq n$ using prefix multiplication. This takes a time of $O(\log n)$ with linear work [25]. In Step 3, we are calculating the values of $C(i,j)$ and

$P(i,j)$ respectively in constant time with n^2 processors.

Later, we are obtaining all the combinations for k elements for $k = 1, 2, \ldots, r$. In Step 5, we are building the array in constant time with n processors which is useful for processor allocation. In Step 6, we are finding the prefix sums of the above array into B_1 in $O(\log r)$ time with $O(r)$ work. In Step 7, the arrays which are needed for processor allocation are formed. These arrays are obtained in constant time with r processors. Note that $p = \sum_{i=1}^{r} 2^i.r = r.2^{r+1}$

In Step 8, generic processor allocation (see Section 2.3) is invoked to get the arrays G and N. By Theorem 2.3 this takes a time of $O(\log\log *(p))$ i.e $O(\log\log *(r.2^{r+1}) = O(\log r)$ with $O(p)$ work on CREW PRAM.

In Step 9, all the needed binary numbers are generated in constant time with p processors. In Step 10, the combination and the complementary combination for each binary number are obtained using prefix sums. Note that the prefix sum for each binary number will go independently and simultaneously with that of others. This step takes a time of $O(\log r)$ with $O(2^{r+1} r \log r)$ work. In Step 11, we are ranking each combination in $O(\log r)$ time. In Step 11, the combinations are stored at the index specified by their rank in constant time with a work of $O(2^{r+1} r \log r)$. Note that $O(2^{r+1} r \log r)$ is $O(P(n,r)r)$.

Now we look at main processor allocation. In Step 12, we formed the array C which is useful for processor allocation. The array stores the information for all the recursive calls that will be made in algorithm. It is based on Eq. (3.11). Of $\log n - 1$ levels of recursive calls, each level has got $min(p,r)$ calls if we are generating permutations for p objects. But the number of arrays we are filling is r for each level for simplicity. From Eq. (3.3), all the extra arrays and extra locations will contain zeroes. So, in processor allocation we won't be allocating processors to them. Thus, the array C is of length $(\log n - 1)r(r+1)$. Filling up C can be done in constant time with $O(\log nr^2)$ processors.

In Step 13, we are calculating the prefix sum of array C into D in $O(\log(\log n.r^2)) = O(\log\log n + \log r)$ time with a work of $O(\log n.r^2)$.

In Step 14, we are forming arrays for doing processor allocation using generic processor allocation. The variable $Total$ is the final summation value in the prefix sums we have calculated above. We later show that this value is bounded above by $2.P(n,r)r$.

The last step follows from the following.

Consider $P(2n,r)/P(n,r)$ when $r \geq 1$.

$$P(2n,r)/P(n,r) = \frac{2n(2n-1)\ldots,(2n-r+1)}{n(n-1)\ldots(n-r+1)} \qquad (3.14)$$

So, $P(2n,r)/P(n,r) \geq 2$.

Coming back to LHS,

$LHS \leq 2P(2n,r)r$

So, in this case our lemma is proved.

Case 2: $r > n$.

$LHS = \sum_{i=0}^{k-1} \sum_{j=0}^{min(2^i,r)} P(2^i,j).j + \sum_{j=0}^{min(n,r)} P(n,j).j = \sum_{i=0}^{k-1} \sum_{j=0}^{2^i} P(2^i,j).j + \sum_{j=0}^{n} P(n,j).j = \sum_{i=0}^{k-1} \sum_{j=0}^{min(2^i,n)} P(2^i,j).j + \sum_{j=0}^{n} P(n,j).j$

Using induction hypothesis and Eq. (3.12),

$LHS \leq 2P(n,n)n + P(2n,n)n \leq 2P(n,n)r + P(2n,n)r$, as $r > n$.

Continuing,

$LHS \leq r(2P(n,n) + P(2n,n)) \leq r(P(2n,n) + P(2n,n))$

This step follows from Eq. (3.14)

So, $LHS \leq 2P(2n,n).r \leq 2P(2n,r).r$, based on Eq. (3.5).

So, our claim is valid in this case also.

Thus, the lemma follows. ∎

We now take up the analysis of our main processing. We generate permutations related to the $\log n$ levels. For the level i, we generate p-permutations of 2^i objects where $1 \leq p \leq min(2^i,r)$. More over we generate the permutations for two such adjacent sets.

Each level takes constant time. The number of operations in the level i are $2[\sum_{j=0}^{min(2^i,r)} P(2^i,j).j]$

We have $\log n$ levels, excluding the top level. Hence the sum of operations of all these levels put together is

$2[\sum_{i=0}^{k-1} \sum_{j=0}^{min(2^i,r)} P(2^i,j).j]$ where $n = 2^k$.

But from Lemma 3.10, we note that the above sum is bounded above by $4P(n,r)r$.

Since there are $\log n$ levels in all each taking constant time. This part takes $O(\log n)$ time.

Now, we have to look at the timing and work details of the top level. We allocate the processors by means of Generic processor allocation problem (see Section 2.3) using Eq. (3.12) or (3.13) whichever is applicable. This is similar to the way we have done processor allocation in the pre-processing.

So, the processor allocation takes a time of $O(\log\log{*n} + \log\log{*r})$ time and the number of operations are $O(P(n,r)r)$. This step runs on the CREW PRAM.

Now, we consider the generation of r-permutations of n objects in the top level. This step takes constant time. The number of operations are $O(P(n,r)r)$.

We summarize in the following theorem.

Theorem 3.11 *The algorithm runs optimally with a time of $O(\log n)$ and the work is $O(P(n,r).r)$ on the CREW PRAM when n is a power of 2.* ∎

3.3.2 Generalization of the Algorithm

In the earlier section, we have seen the algorithm to generate r-permutations of n objects when n is a power of 2. We will remove that restriction in this section. The algorithm given in this section works only when n is *not* a power of 2. It heavily makes use of our earlier restricted algorithm.

Like our earlier algorithm, this algorithm is also based on special cases of Vandermonde's convolution.

Define $x = \lceil \frac{n}{2} \rceil$ and $y = \lfloor \frac{n}{2} \rfloor$

Consider the following application of Vandermonde's convolution.

$$\sum_{k=0}^{r} P(x,k)C(r,k)P(y,r-k) = P(x+y,r) = P(n,r) \tag{3.15}$$

So, we would like to generate the r-permutations of n objects using the permutations of x objects $\{1,2,\ldots,x\}$ and that of y objects $\{x+1, x+2, \ldots, n\}$. In the above equation we eliminate the zero terms considering r with respect to x and y

Consider,

$$\sum_{k=0}^{r} P(x,k)C(r,k)P(y,r-k) = P(n,r) \tag{3.16}$$

if $r \leq y$.

$$\sum_{k=0}^{y} P(y,k)C(r,k)P(x,r-k) = P(n,r) \tag{3.17}$$

if $r > y$ and $r \leq x$.

$$\sum_{k=r-x}^{y} P(y,k)C(r,k)P(x,r-k) = P(n,r) \tag{3.18}$$

if $r > x$.

As $x = \lceil \frac{n}{2} \rceil$ and $y = \lfloor \frac{n}{2} \rfloor$ observe

Observation:

1. $x \geq y$
2. $x = y + 1$ if n is odd
3. $x = y$ if n is even
4. $x + y = n$

So, the idea behind our algorithm is that we generate the 'needed' permutations of x objects and similarly the needed permutations for y objects and combine them using above applicable equation to generate the r-permutations of n objects. By 'needed' permutations we mean the permutations that are needed in the above equations depending on the case which is applicable. For example, if $r > x$ then we need to generate the k-permutations of x objects for $r - y \leq k \leq x$ and k-permutations of y objects for $r - x \leq k \leq y$. But in any case, the following observation is quite useful throughout our algorithm and its proof of correctness. It follows from the above equations.

Observation 4: *In our algorithm, whenever we generate p-permutations for some set of objects of cardinality q then the following properties are obeyed.*

1. $p \leq q$

2. $p \leq r$

3. $p \geq 0$

We show how we go about generating the needed permutations for x objects and that of y objects.

We use the following abbreviations in our presentation.

We "*copy* k-permutations of a objects where for all k, $l_1 \leq k \leq l_2$ by adding e to each element" for the following procedure.

We consider copying the k-permutations of a objects by adding e to each element for a particular value of k as a task; we have $l_2 - l_1 + 1$ tasks. The i-th task involves copying of the $l_1 + i - 1$-permutations. Obviously, the i-th task needs $P(a, l_1 + i - 1).l_1 + i - 1$ processors. We make $A[i] = P(a, l_1 + i - 1)(l_1 + i - 1)$ for each i. Then we apply prefix sums on array A in $O(\log(l_2 - l_1 + 1))$ time with $O(l_2 - l_1 + 1)$ work. Using this information, we apply our generic processor allocation to allocate processors. Let us denote by Sum the expression $\sum_{i=l_1}^{l_2} P(a, i).i$

For generic processor allocation, the time taken is $O(\log\log*(Sum))$ with $O(Sum)$ work on CREW PRAM. So, the total time taken is $O(\log\log*(Sum) + \log(l_2 - l_1 + 1))$ with $O(Sum + l_2 - l_1 + 1)$ work on CREW PRAM.

By "*processor allocation* for k-permutations of a objects where for all k, $l_1 \leq k \leq l_2$" we mean the following procedure.

We would like to generate the k-permutations of a objects using Vandermonde's convolution; the special case specified as part of the above statement. So, we need to allocate the processors for these $l = l_1 - l_2 + 1$ calls. For this, we first form an array C of length $l(l_2 + 1)$ such that the locations $(i - 1)(l_2 + 1) + 1$ to $i(l_2 + 1)$ are for i^{th} call. We fill the locations using the Vandermonde's' convolution and note that the extra locations we are filling will contain zeroes. (Since for generating i-permutations we need at most $i + 1$ locations in Vandermonde's convolution) . After that we apply prefix sum on this array and then apply generic processor allocation to get the data structure for processor allocation.

Keeping this notation in view the algorithm given below specifies formally generation of r-permutations of n objects.

Pre-processing

Calculate i! for $0 \leq i \leq n$.

Calculate C(i,j)'s, P(i,j)'s for $0 \leq i,j \leq n$.

Generate the i-combinations of j objects for all $1 \leq i,j \leq r$.

Algorithm Generate_Permutations_generalized(n,r)

Begin

$x = \lceil \frac{n}{2} \rceil$; $y = \lfloor \frac{n}{2} \rfloor$;

// Generate the permutations for x objects

x = a + b where a is max power of 2 such that a < x. ($a = 2^t$)

Processor allocation for the generation of k-permutations of 2^i objects where k varies from 1 to $min(2^i, r)$ and i varies from 0 to t.

Generate k-permutations of 2^i objects where k varies from 1 to $min(2^i, r)$ and i varies from 0 to t.

Copy k-permutations of a objects for $1 \leq k \leq min(b,r)$
adding a to each element.

Identify all elements greater than x as a dummy element and find for each permutation whether it contains dummy elements or not.

Rank all the permutations which do not have dummy element. We have got the k-permutations of b objects where k varies from 1 to min(b,r).

Processor allocation for generation of permutations of x objects.

Generate the permutations of x objects using the permutations of a objects and that of b objects.

```
// Generate the permutations of y objects

      Copy the k-permutations of a objects where k varies from 1
      to min(a,r) by adding x to each element.

      Generate the k-permutations of b-1 objects using the above
      generated permutations using ranking.

      Processor allocation for generation of permutations of y objects.

      Generate the permutations of y objects using the
      copied permutations of a objects and that of b-1 objects.

// Generate the r-permutations of n objects
      Generate the r-permutations of n objects using the
      permutations of x objects and that of y objects.
End.
```

We first explain the pre-processing part of the algorithm. In the pre-processing we find (a) $i!$, (b) $C(i.j)$'s and (c) $P(i.j)$'s for $0 \leq i,j \leq n$. Later we are generating the i-combinations of j objects for $0 \leq i,j \leq r$. These are already explained in our earlier algorithm. From that algorithm we understand that, the pre-processing can be accomplished in $O(\log n)$ time with $O(n^2 + 2^{r+1} r \log r)$ work.

Now, we look at the generation of permutations for x objects and the permutations for y objects. We first look at the generation of permutations for x objects. We indicate how it can be modified to generate the permutations for y objects later, which is similar.

Generating Permutations for x objects $\{1,2,\ldots,x\}$

We express $x = a + b$ where a is defined as the largest power of 2 such that $a < x$. Let $a = 2^t$ for some integer t. The following observation is not difficult to make.

Observation:

1. $a < x$

2. $a \geq b$

3. $b > 0$

Note that if x is a power of 2 then $a = b$.

The permutations for x objects we will obtain by using the permutations of a objects $\{1, 2, \ldots, a\}$ and the permutations of b objects $\{a+1, \ldots, x\}$. That is, we are using the following application of Vandermonde's convolution for generation of k-permutations of x objects.

$$\sum_{i=0}^{k} P(a,i)C(k,i)P(b,k-i) \tag{3.19}$$

We proceed as follows. We initially generate the k-permutations of a objects where k varies from 1 to $min(a,r)$. Later we generate the k-permutations of b objects where k varies from 1 to $min(b,r)$. Note that the length of permutations i,e, k, we need from these a objects and b objects will not be outside this range. This follows because if it is outside then either the length exceeds r (which is not possible; see Observation 4) or length exceeds the number of objects in the set, in which case the number of permutations are zero (which does not arise as we have specifically avoided zero terms in our identities).

Now, we look at the generation of permutations for a objects.

Generation of Permutations for a objects $\{1, 2, \ldots, a\}$

Recall that in our earlier algorithm, if we are generating the $min(a,r)$-permutations for $2a$ objects, (note that $2a$ is a power of 2), then we first generate the k-permutations for 2^i objects where k varies from 1 to $min(2^i, r)$ and i varies from 0 to t (Recall $a = 2^t$). Later on we generate the r-permutations of $2a$ objects. The time taken is $O(\log a)$.

In the present case, we generate the k-permutations for 2^i objects where k varies from 1 to $min(2^i, r)$ and i varies from 0 to t. This involves processor allocation followed by the generation of permutations. But processor allocation and the generation have same resource bound for work. The total work for the generation of permutations of a objects is:

$W = \sum_{i=0}^{t} \sum_{j=0}^{min(2^i,r)} P(2^i, j).j$

To bound this sum, we consider two cases.

Case 1: $r \leq 2a$

Using the Lemma 3.10, the above sum can be bounded as follows.

$W \leq 2.P(2a, r).r \leq 2.P(n, r).r$

since $a < x, 2a \leq n$.

Case 2: $r > 2a$

$W = \sum_{i=0}^{t} \sum_{j=0}^{min(2^i, r)} P(2^i, j).j = \sum_{i=0}^{t} \sum_{j=0}^{2^i} P(2^i, j).j = \sum_{i=0}^{t} \sum_{j=0}^{min(2^i, 2a)} P(2^i, j).j \leq 2P(2a, 2a)2a$

In the last step we used Lemma 3.10. Further,

$W \leq 2.P(2a, 2a).r \leq 2.P(n, 2a).r \leq 2.P(n, r).r$

The second step follows because $r > 2a$ and the third step follows because $n \geq 2a$ and the last step follows because $2a < r \leq n$ and using Eq. (3.5).

Thus, generation of permutations of a objects is bounded by $O(P(n, r).r)$. The time taken is $O(\log a)$.

So, we have with us the k-permutations of a objects $\{1, 2, \ldots, a\}$ for all k, $1 \leq k \leq min(a, r)$. Using them we generate the k-permutations of b objects $\{a+1, \ldots, x\}$ for, $1 \leq k \leq min(b, r)$.

Generation of Permutations for b objects $\{a+1, a+2, \ldots, x\}$

We first copy the k-permutations of a objects adding a to each element where k varies from 1 to $min(b, r)$. (Recall that $a \geq b$.) Now, we have with us the k-permutations of a objects $\{a+1, a+2, \ldots, 2a\}$ where k varies from 1 to $min(b, r)$. From these we would like to generate the k-permutations of b objects $\{a+1, a+2, \ldots, x\}$. So, if $a = b$, then since $a + b = x$, it follows that $2a = x$ and we are done.

On the other hand, if a is strictly greater than b, then the following approach is used. If we have with us the k-permutations of a objects $\{a+1, \ldots, 2a\}$, it contains all the k-permutations of b objects $\{a+1, a+2, \ldots, x\}$ one and only once since $x < 2a$. So, the remaining k-permutations contain at least one element from the set $\{x+1, x+2 \ldots, 2a\}$ (from now onwards we call them *dummy* elements).

So, we assign k processors to each k-permutation of a objects $\{a+1, a+2 \ldots, 2a\}$ where k varies from 1 to $min(b, r)$. In the first step these k processors are assigned

ı such a way that one processor is assigned to one element of the permutation and it hecks whether the element it is looking at is a dummy element or not. If it is it puts a in a corresponding location else puts a 0 in it. After this step, we find the OR of these its for each permutation independently and simultaneously. At the end of this step, we 'ill know for each permutation whether it contains a dummy element or not.

So, the remaining work is to bring together the non-dummy permutations (those that oes not contain any dummy elements). We *rank* each non-dummy permutation to get s index among the non-dummy permutations. We can rank a k-permutation using k rocessors in $O(\log k)$ time on CREW PRAM (see Section 2.2.1). We use this procedure) first rank the permutations (non-dummy) and put them at the position indexed by their ank. The ranking of all the permutations goes independently and simultaneously. This :ep takes a time of $O(\log(min(b,r)) = O(\log n)$. Note that while ranking we will ssign processors to *all* permutations (not necessarily non-dummy permutations), but ıose processors assigned to dummy-permutations keep idle.

So, the work done is

$W = \sum_{i=0}^{min(b,r)} P(a,i).i \log i$

To bound this sum, we consider two cases.

Case 1: $b \geq r$:

$W = \sum_{i=0}^{min(b,r)} P(a,i).i \log i = \sum_{i=0}^{r} P(a,i).i \log i \leq \sum_{i=0}^{r} P(a,i).r \log r = r \log r. \sum_{i=0}^{r} P(a,i)$

Consider the following special case of Vandermonde's convolution.

$P(x,r) = \sum_{i=0}^{r} P(a,i).C(r,i).P(b,r-i)$

From this it follows that $\sum_{i=0}^{r} P(a,i) \leq P(x,r)$

So, $W \leq r \log r P(x,r)$

The following lemma is useful later in the proof.

nma 3.12 *if $x = \lceil \frac{n}{2} \rceil$ and $r \leq x$, then $P(n,r)/P(x,r) \geq 2^{r-1}$.*

of: Consider, $\frac{P(n,r)}{P(x,r)} = \frac{n}{x}\frac{n-1}{x-1} \cdots \frac{n-r+1}{x-r+1}$

There are r terms in each of numerator and denominator.

Ve consider two cases

Case 1: n is even:

$x = \frac{n}{2}$ or $2x = n$.

$2(x-i) \leq n-i$ if $0 \leq i$.

It follows that $\frac{P(n,r)}{P(x,r)} \geq 2^r$ or $\frac{P(n,r)}{P(x,r)} \geq 2^{r-1}$.

Case 2: n is odd:

$x = (n+1)/2$ or $n = 2x-1$

So, $2(x-i) \leq n-i$ for $i \geq 1$

Using this it follows that $\frac{P(n,r)}{P(x,r)} \geq 2^{r-1}$.

Thus, our claim is verified. ∎

Using Lemma 3.12, we have $P(x,r) \leq P(n,r)/2^{r-1}$

So, $W \leq r \log r P(x,r) \leq r \log r P(n,r)/2^{r-1} \leq 2rP(n,r)$,

as $2^r \geq \log r$.

So, W is bounded by $O(P(n,r).r)$.

Case 2: $b < r$:

$W = \sum_{i=0}^{min(b,r)} P(a,i).i \log i = \sum_{i=0}^{b} P(a,i).i \log i$

$\leq \sum_{i=0}^{b} P(a,i).b \log b = b \log b. \sum_{i=0}^{b} P(a,i)$

Consider the following special case of Vandermonde's convolution.

$P(x,b) = \sum_{i=0}^{b} P(a,i).C(b,i).P(b,b-i)$

It follows that $\sum_{i=0}^{b} P(a,i) \leq P(x,b)$

So, $W \leq b \log b P(x,b)$

Using Lemma 3.12, we have $P(x,b) \leq P(n,b)/2^{b-1}$

So, $W \leq b \log b.P(x,b) \leq b \log b P(n,b)/2^{b-1} \leq 2bP(n,b)$,

as $2^b \geq \log b$.

So, $W \leq 2.rP(n,b) \leq 2.rP(n,r)$

The first step follows because $b < r$ and the next step follows because $b < r \leq n$.

Thus, it follows that the work done for getting the k-permutations of b objects $\{a+1, a+2 \ldots, x\}$ is bounded by $O(P(n,r).r)$ with $O(\log n)$ time complexity.

Now that we have with us all the permutations needed for the a objects $\{1, 2, \ldots, a\}$ and b objects $\{a+1, \ldots, x\}$. So, the next step is to generate the needed permutations for x objects $\{1, 2, \ldots, x\}$ by combining these permutations. But the lengths of permutations that we need depends upon the value of r with respect to x and y. In other words, we need to know which of the Eqs. (3.16), (3.17), (3.18) is applicable, then we will know the

range of lengths for the permutations needed for x objects. For example, if Eq. (3.17) is applicable, then range is $r - y$ to r.

So, depending on the equations's applicability, we do the processor allocation for generating permutations of x objects for that range using Eq. (3.19). We then generate the permutations for the corresponding range. The time taken is $O(\log n)$ with work bounded by $O(P(n,r).r)$ as we are generating only the needed permutations and depending upon the case we use Eqs. (3.16), (3.17), (3.18) whichever is applicable to bound the work. So, at the end of this step, we have generated needed permutations for x objects.

Generation of permutations for y objects $\{x+1, x+2, \ldots, n\}$

Now, in a similar manner, we have to generate the permutations for y objects, where the range of permutations length is specified by the applicability of Eqs. (3.16), (3.17), (3.18). For example, if Eq. (3.18) is applicable then the range is $r - x$ to y.

If n is even then $x = y$. Then obviously, the length of permutations needed from both the x objects and y objects is same (see Eqs. (3.16), (3.18) and Eq. (3.17) will not be applicable in this case). So, we simply copy all the k-permutations of x objects except we add x to each element.

On the other hand, if n is odd, then we proceed as follows.

If n is odd, $x = y + 1$. Since, $x = a + b$, it follows that $y = a + (b - 1)$.

So, our idea is, like the earlier approach, we generate the k-permutations of a objects for $1 \leq k \leq min(a, r)$ and k-permutations of $b - 1$ objects for $1 \leq k \leq min(b - 1, r)$ and combine them to get the needed permutations of y objects. Note that, if $b = 1$, then this last step is not needed and the permutations from a objects itself serve as the permutations for y objects.

We proceed in a similar manner to generate the needed permutations of y objects in $O(\log n)$ time with $O(P(n,r).r)$ work.

Once we have with us the needed permutations of x objects and y objects, we use Eqs. (3.16),(3.17),(3.18) whichever is applicable. We first allocate the processors based on this equation and later generate the r-permutations of n objects. The time taken is $O(\log n)$ (for processor allocation) and work done is $O(P(n,r).r)$.

We obtain the following theorem.

Theorem 3.13 *The r-permutations of n objects can be generated in $O(\log n)$ time with*

optimal work $O(P(n,r).r)$ on CREW PRAM. ∎

3.4 Conclusions

We have discussed two algorithms for generating r-permutations of n objects. Both are optimal and work on the CREW PRAM. While the r-recursive algorithm generates the permutations lexicographically, the n-recursive algorithm does not.

Chapter 4

Efficient Parallel Algorithms for Generation of Permutations

4.1 Introduction

In this chapter we discuss parallel algorithms for enumerating the permutations. (For introductory comments refer to Chapter 3)

This chapter is organized as follows. In Section 2, we present a cost optimal algorithm to generate r-permutations with time complexity $O(r)$. This algorithm is based on the concept of permutation tree. In Section 3, we present an efficient algorithm, which runs in $O(\log n)$ time but generates the permutations in the restricted case where we want to generate the $n!$ n-permutations of n objects. In Section 4, we generalize this algorithm to generate all the r-permutations of n objects in $O(\log n)$ time. In Section 5, we indicate the connection between our improved algorithm and a restricted version of parallel integer sorting; any improvement in algorithm for integer sorting will result in corresponding improvement in our algorithm. Some conclusions are offered in Section 6.

4.2 Basic Algorithm

Our first algorithm to generate permutations is based on a data structure which we call *permutation tree*. A permutation tree captures the notion of generation of all the permutations in a succinct and simple manner. Similar tree structures are used by [7] to

represent the enumeration of all the possible paths in a graph, there they are called as *logic trees*; we believe "permutation tree" is a more natural term for this data structure.

4.2.1 Permutation Tree

We assume that the elements of S can be ordered and we are interested in generating r-permutations out of n elements from S. Let $S[i] = i$ be the i th element of set S.
Definition: *An* (n, r, S)–Permutation Tree *corresponding to the* r*-permutations of* n *objects present in* S *is a tree with following properties:*

1. *If* $r = 1$ *then the root has* n *children and* ith *child is labeled with* $S[i]$.
2. *If* $r > 1$ *then the root has* n *labeled children such that the label of* v_i*, the* ith *child, is* $S[i]$ *and the* i*th subtree rooted at* v_i *is an* $(n-1, r-1, S-\{S[i]\})$ *permutation tree.*

It is easy to observe that, each subtree of an (n, r, S) permutation tree is a permutation tree. The path from any child of root to a leaf is an r-permutation of n objects and we call this the *permutation associated* with that leaf. Permutations associated with different leaves are distinct. Furthermore, the permutation tree preserves the lexicographic order of the permutations, in the sense that if a leaf v is to the *left* of another leaf w, then the permutation corresponding to the first leaf v lexicographically precedes the permutation corresponding to the leaf w. This property can be easily proven by induction on the number of levels of the permutation tree.

By a *path* we mean the concatenation of labels associated with the nodes in order. A leaf p is *left* of a leaf q if and only if there exist nodes s, t, u such that p is a descendant of t, q is a descendant of u, t and u are children of s, and t occurs before u. For the relevant graph theory terminology refer to [14]. An example of a permutation tree for the case where $n = 3$ and $r = 2$ is given in Figure 1.

4.2.2 Algorithm

Our algorithm builds a permutation tree initially and then lists all root to leaf paths to generate the required permutations. The algorithm works as follows.

The list of n positive integers $S = \{1, 2, \ldots, n\}$ are stored in an array A (i.e., $A[i] = i$) and we are interested in generating m-permutations of elements from S. The

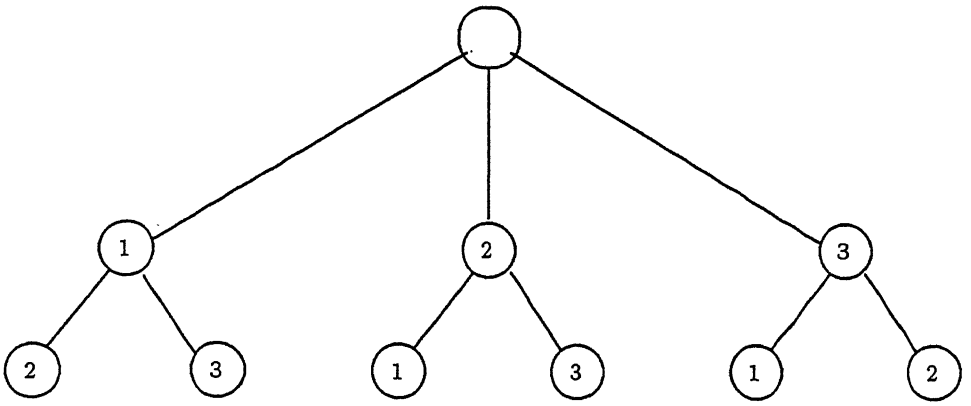


Figure 1: Permutation tree when $n = 3, r = 2$

root has n children and we associate $n-1$ processors with each child. The first processor in the set of processors associated with ith child copies $A[i]$, the ith element from the array into the label associated with that child. In the next step, for each child v, the list of processors associated with node v copy all the values from the array A except the value stored in that child to an array associated with that node. Now, for each child, this algorithm is applied recursively (considering that child as the root). This recursive call correspond to generating $(m-1)$-permutations from $n-1$ elements from S which are associated with that particular node. The algorithm terminates when $m = 1$ in which case, only the first step is sufficient and no copying of remaining elements from the array to a different array takes place.

Once the tree is built, we associate a processor with each leaf and it reads the path to get the permutation. A number is associated with each processor which represents the *rank* of that permutation in the lexicographic order.

The algorithm is formally given in Fig. 2; we depend upon the indentation to show the boundaries of statements.

By $A - A[i]$ we mean the array formed by removing the ith element of A. This can be obtained by $n-1$ processors in constant time, if A contains n elements. The last step is a parallel recursive call. In this algorithm when B is a two dimensional array then, $B[i]$, refers to the ith row of B as a whole. The fifth parameter of *Gen_Perm* is specifically provided for the purpose of processor allocation. For the call *Gen_Perm(node,* A, n, m, a*)*, processors a to $a + P(n,m) - 1$ should be allocated.

```
Algorithm Generate_Permutations(n,m)
Begin
      Create_node(Root);
      for i = 1 to n in parallel do
          A[i] = i;
      Gen_Perm(Root, A,n,m, 1);
End;

Algorithm Gen_Perm(node,A,n,m,a)
Begin
     Create 'n' nodes, Node[1..n];
     /* Copy label */
     for i = a to a+n-1 in parallel do
         Node[i-a+1].parent = node;
         Node[i-a+1].label =A[i-a+1];
         if (m==1)
            /* Generate permutations */
            Write the reversal of path from node[i] to the child of the root as
            the permutation in location 'i'.
     if (m > 1 ) then
        for i = 1 to n in parallel do
            B[i] = A - A[i] /* B[i] contains all the elements of A except its
            i-th element.*/
        for i = 1 to n in parallel do
            Gen_Perm(Node[i], B[i],n-1, m-1, a + (i - 1)P(n - 1, m - 1));
End.
```

Figure 2: Algorithm 1

The algorithm is building the tree level by level. As building a single level takes constant time, we have the following theorem.

Theorem 4.1 *All $P(n,m)$ permutations can be generated lexicographically in $O(m)$ time with $O(P(n,m))$ processors on the CREW model.*

Proof: We will prove the theorem by mathematical induction. We first prove that time complexity is $O(m)$ by induction on m. When $m = 1$, the algorithm takes only one step. Let us assume the algorithm for $m \leq k$ takes atmost cm time (for some constant c).

Now consider the case where $m = k + 1$, In the algorithm (after taking a constant number of steps), there are n recursive calls. All of these execute simultaneously. Hence the time complexity is $T(k) + O(1) \leq c_1 + ck \leq c(k+1) = cm$, if $c_1 \leq c$. Hence our first claim is validated.

We next show again by induction on m that processor complexity is $O(P(n,m))$. For $m = 1$, clearly the number of processors needed are $n = P(n,1)$.

Let us assume the claim is valid for $m \leq k$.

For the case where $m = k + 1$, there are n recursive calls to Gen_Perm and by induction hypothesis each of the calls needs $P(n-1,m-1)$ processors. Hence, the number of processors needed are $\max(n, n(n-1), nP(n-1,m-1)) = \max(n, n(n-1), P(n,m)) = P(n,m)$, since $P(n,m) = nP(n-1,m-1)$. Hence our claim is validated.

Further as there are no simultaneous writes, the algorithm can be executed on the CREW model. As permutation tree stores the permutations in lexicographic order, the theorem follows. ∎

As each m-permutation is of length m and as total number of permutations is $O(P(n,m)$, for permutation generation $\Omega(P(n,m)m)$ is a lower bound. The *cost* of our algorithm is $O(mP(n,m))$. Hence the algorithm is cost optimal.

4.3 Logarithmic Time Algorithm for n-permutations of n objects

In this section we discuss a more efficient algorithm to generate permutations for a restricted version. We generalize this algorithm in the next section.

The problem we are concerned in this section is formally defined as follows:

```
Algorithm Generate_permutations(n)
Begin
      for i = 1 to n!  in parallel do
          for j = 1 to n do perm[i,j] = j;
          k[i] = i-1;
          m[i] = (n-1)!;
          for j = n-1 downto 1 do
              /* Get factorial representation */
              a[i] = k[i] div m[i];
              k[i] = k[i] mod m[i];
              m[i] = m[i]/j;
              swap(perm[i,j+1], perm[i,j+1-a[i]]);
End.
```

Figure 3: Algorithm 2

Restricted Problem: *Generate all n-permutations of n objects.*

We have to generate all the $n!$ permutations each of length n. Our algorithm is based on a linear time algorithm given in [34]. We reduce the time required to $O(\log n)$. Algorithm uses the concept of factorial representation[28] of numbers in the range 0 to $n! - 1$.

Lemma 4.2 *Any natural number k from 0 to $n! - 1$ can be uniquely represented as a sequence $(a_{n-1}, a_{n-2}, \ldots, a_1)$ with $0 \leq a_i \leq i$ such that $k = \sum_{i=1}^{n-1} a_i i!$*

In [28] there are number of algorithms which establish a correspondence between the above representation and generation of permutations. The algorithm of [34], given in Fig. 3 exploits this correspondence.

In the above algorithm *swap(perm[i,p],perm[i,q])* swaps corresponding elements of the two-dimensional array *perm*. When the algorithm terminates, *perm[i]* contains the

permutation corresponding to the integer $i-1$. But unfortunately, the algorithm does not generate permutations in lexicographic order.

We next give a proof of the correctness of algorithm; the original paper did not provide the proof. Moreover, we use some of the arguments from the proof in our improved algorithm.

Theorem 4.3 Algorithm 2 *correctly generates the n-permutations of n objects.*

Proof: Observe that in the algorithm the only operations that are applied on the output array, *perm* are swaps. More over, initially the numbers $1 \ldots n$ are copied into each array. Hence when the algorithm terminates, each array certainly contains a permutation of n objects. As we are generating $n!$ permutations, to complete the proof it is sufficient to show that permutations that are generated are *distinct.*

We use induction. Consider the case, where $n = 1$, the array will contain only the trivial permutation (1) and from the algorithm we can see that the algorithm correctly generates this permutation. So our basis is proved.

By induction hypothesis let us assume that, the algorithm works correctly for the case when $n \leq k$. Now, consider the case when $n = k+1$. We prove by contradiction. Let us assume (for sake of contradiction) that the permutations generated are not all distinct. So there must exist two integers p and q for which the algorithm generates the same permutation. Let us consider the factorial representations of p and q (see Lemma 4.2). Let us assume

$p = (s_{n-1}, s_{n-2}, \ldots, s_1)$

$q = (t_{n-1}, t_{n-2}, \ldots, t_1)$.

Now *perm*[p] is the permutation corresponding to integer p and *perm*[q] is the permutation corresponding to integer q. Based on our assumption, *perm*$[p, i]$ =*perm*$[q, i]$, $1 \leq i \leq n$. This implies that *perm*$[p, n]$ = *perm*$[q, n]$. In Algorithm 2, the $perm[p, n]$ participates in only one swap, swap(*perm*$[p, n]$,*perm*$[p, k_1]$) where $k_1 = n - s_{n-1}$. Similarly, the corresponding swap for $perm[q, n]$ is swap(*perm*$[q, n]$,*perm*$[q, k_2]$) where $k_2 = n - t_{n-1}$. Moreover, this is the first swap that takes place when the swapping starts for each permutation; but since *perm*$[p, n]$ =*perm*$[q, n]$, it implies *perm*$[p, k_1]$ =*perm*$[q, k_2]$. But when first swap takes place *perm*$[p, k_1]$ contains k_1 and *perm*$[q, k_2]$, contains k_2, thus $k_1 = k_2$, or $n - s_{n-1} = n - t_{n-1}$ and hence $s_{n-1} = t_{n-1}$.

Let $p' = (s_{n-2}, \ldots, s_1)$ and $q' = (t_{n-2}, \ldots, t_1)$. As $p \neq q$, it follows that $p' \neq q'$. The k_1th ($k_1 = k_2$) position of *perm*$[p]$ and *perm*$[q]$ have value n, From Lemma 1, p' and q' are valid integers with respect to $n-1$ permutations, thus after finding the permutation corresponding to p' and q', they will be identical. But this contradicts our induction hypothesis. Hence our proof is complete. ∎

4.3.1 Basic Idea

Now we discuss a more efficient parallel implementation of the above algorithm.

First, we find all the swaps that will be taking place. Then, we simulate the swaps. To find swaps we have to get the factorial representation of an arbitrary integer in the range $[0..n! - 1]$. For this we give a much faster algorithm (Step 1 of Algorithm 4). We do this in constant time. Finding the result of all swaps is non-trivial. Here we observe that after all swaps are over, each position of the array will contain some number. For example, suppose *perm*$[a, 2]$ has the value 6. What this means is that through some swaps, *perm*$[a, 2]$ has got the value *perm*$[a, 6]$. This might have taken more than one step. There can be a chain in the sense that *perm*$[a, 4]$ might have got *perm*$[a, 6]$ and *perm*$[a, 2]$ might have been interchanged with *perm*$[a, 4]$. So, in other words we are interested in knowing what is the final value that will enter into a particular location without *actually* performing all these swaps.

We first describe some notation which will be used later. Ordered pair (p, q) when $p \leq q$ will denote swap between *perm*$[a, p]$ and *perm*$[a, q]$ for an arbitrary permutation *perm*$[a]$. Moreover we say that q is *right paired* with p and p is *left paired* with q. Similarly if (p, q) is a swap then we say p is in the *left position* and q is in the *right position*.

Observe that each index p is right paired with exactly one index and that is *last* swap in which index p participates. More over p may be left paired with any number of indices (possibly zero). But if there are two swaps (p, a_1) and (p, a_2) with $a_1 < a_2$, then the first swap takes place after the second swap has taken place. Further, if (p, a_2) is *the* swap that has taken place *just before* (p, a_1) with p in the left position, then the final value that *comes* into position of a_1 depends on the value in a_2, when it is swapped with p. But this depends upon the final swap in which a_2 is in the left position. So to facilitate this case we maintain a linked list. For each element in the range $1 \ldots n$ there is a node.

A node corresponding to p points to q if and only if there is a swap (p, q) and it will be the last swap that has taken place with p in the left position in Algorithm 2. Or, in other words, q is the *smallest* of all the elements that have participated in swaps with p in left position. There may be a number of linked lists but the node corresponding to an element will be in exactly only one linked list; thus the total number of linked lists is atmost n, *i.e*, the number of nodes.

Let us consider the case, where, (p, a_1) is the swap that immediately takes place after the swap (p, a_2) has taken place *with p in the left position.* As each element participates in only one swap in right position, it follows that, the final value that comes into the position a_1 depends upon this swap (p, a_1) only. But the value that comes into a_1 is the value that has *finally* come into a_2 before it participated in a swap with p. If node corresponding to a_2 points to some node say, b_1, then whatever the final value that has come into b_1 finally will be the element that comes into a_2 before its swap. This recursive structure terminates when a node, say, v' has no next node; or equivalently, the index corresponding to node v', say, v, has no swap with v in the left position. So, $v'.next = nil$ as there is no swap such that v is in left position. Thus with respect to linked lists, for each node, it is the element or value corresponding to the node which is the last node in that linked list, which will be the final value that enters into that node when it participates in its swap in right position (if any).

Now consider possibility of $swap(p, p)$. This happens when the digit corresponding to the element p in the factorial representation for that particular permutation is zero. What this in effect means is this is no swap at all. But we know that this is the only swap that takes place with p in the right position. So, if we want to know what is the final value that ends up in the pth position, then p will be head of a linked list and more over, the element corresponding to the last node in that linked list will be the element which will finally reside in position p. Similar is the case for the first element of the permutation which has no associated swap; in the factorial representation we have only one $n - 1$ digits for a number and they are (respectively) associated with indices $2 \ldots n$ of the permutation. As a special case, if (p, q) is the *first* swap with p in the left position then the final value that will go to position of q is p. These observations are used in Algorithm 3 (see Figure 4).

```
Algorithm Gen_all_perm(n)
Begin
  for j = 0 to n!-1 in parallel do
1:    for i = 1 to n-1 in parallel do
          a[j,i] = (j mod (i+1)!  )div i!  ;
          b[j,i+1].(key,data) = (i+1 - a[j,i], i+1);
2:    sort b[j]) in stable manner
      /* Finding the minimums */
3:    for i = 1 to n in parallel do
3.1:      c[j,i].min = 0;
3.2:      Assign C[j,i].min = p, in such a way that 'p' is the minimum
          element that took part in a swap with element 'i' for the
          permutation corresponding to integer 'j'.
          /*In array b[j] if two adjacent
          elements are (a1,a2) and (b1,b2) and if (a1 <> b1) then C[j,b1] = b2.  */

          /* Set the linked list*/
3.3:      if c[j,i].min != 0 then d[j,i].next = c[j,i].min;
          else d[j,i].next = nil;
4:    Compress(d[j]);
5:    for i := 1 to n in parallel do
          if (i==1) then perm[j,1] = d[j,1].next
          /* First element has no associated digit in factorial representation */
          else
               if (b[j,i].key != b[j,i+1].key then
               /* b[j,i].data is the first index which
               swaps with index b[j,i].key */
                  perm[j,b[j,i].data] = b[j,i].key ;
               else
                  perm[j,b[j,i].data] = d[j,b[j,i+1].data].next ;
End.
```

Figure 4: Algorithm 3

In Algorithm 3, we want to sort $b[j,2]\ldots b[j,n]$ for an arbitrary jth permutation; the first element $b[j,1]$ is excluded. More over, we assume the sorting algorithm to be *stable*. In Step 1 we are getting the factorial representation of a number; this step can be done in constant time with n processors. (provided division is part of instruction set and word size is sufficiently large). Step 2 involves sorting which can be done in $O(\log n)$ time with n processors[12]. Step 3.1 can be done in constant time As sorting in Step 2 is stable, Step 3.2 can be done in constant time with n processors using the information available in the array 'b'. It is easy to observe that Step 3.3 can also be done in constant time. So in other words Step 3 can be done in constant time with $O(n)$ processors.

In Step 4, we have a set of linked lists. There will be an associated node for each integer in the set $S = \{1,\ldots,n\}$. More over each node appears in only one linked list. In other words the sum of length of all these linked lists is n. Compressing these linked lists means, all nodes must point to their last node in their linked list. Further if for some arbitrary node v if *v.next = nil*, then after compressing *v* will point to itself. This can be done in $O(\log n)$ time with $O(\frac{n}{\log n})$ processors[25]. It is easy to observe that Step 5 can be done in constant time with $O(n)$ processors. All these steps will be done in parallel for $n!$ integers in the range 0 to $n! - 1$.

In Step 1 we are simultaneously using (or reading) the value of "j" for getting the factorial representation of the number. This step is executing in constant time with n processors. Suppose if we broadcast this value of j on an EREW model to all the n processors in $O(\log n)$ time with $\frac{n}{\log n}$ processors, then Step 1 will take $O(\log n)$ with $\frac{n}{\log n}$ processors on an EREW model. So, in essence as we can clearly see, all other steps can be made to execute on an EREW model with the same time and processor complexities (note that we are assuming that n is known to every processor at the beginning of execution itself). Hence we can execute this algorithm on an EREW model. Summarizing all these observations leads to the following theorem.

Theorem 4.4 *We can generate all $n!$ permutations of n numbers taken all at a time in $O(\log n)$ time with $O(n!n)$ processors on an EREW model.* ∎

The processor time product of this algorithm, is $O(n!n\log n)$, which is not cost optimal. We can see that except for sorting, all the remaining steps can be done with a time complexity of $O(\log n)$ and with a processor complexity of $O(\frac{n}{\log n})$. In the above

algorithm, we have used merge sort to sort items which are integers in the range $1 \ldots n$. But we can use more efficient parallel integer sorting algorithms. But unfortunately, there is no known optimal $O(\log n)$ time parallel deterministic integer sorting algorithm on $O(\log n)$ bit word sized machines. If we use the Bhatt *et al's* [9] parallel integer sorting in place of the above parallel merge sort then this leads to the following theorem.

Theorem 4.5 *We can generate all $n!$ permutations of n numbers taken all at a time in $O(\log n)$ time with $O(n \log \log nn!)$ work on the ARBITRARY CRCW PRAM.* ∎

We explore the relationship between parallel integer sorting and our algorithm more fully in Section 5. However, there are randomized [35] and non-conservative [24] algorithms for integer sorting which achieve the cost optimality. Thus we have following theorems:

Theorem 4.6 *We can generate all $n!$ permutations of n numbers taken all at a time in $O(\log n)$ time with optimal $O(n!\frac{n}{\log n})$ processors using a randomized algorithm on EREW PRAM.* ∎

Theorem 4.7 *We can generate all $n!$ permutations of n numbers taken all at a time in $O(\log n)$ time with $O(n!\frac{n}{\log n})$ processors on an EREW PRAM having a word length $\Omega(\log^2 n)$.* ∎

4.4 Logarithmic Time Algorithm

In Section 3, we solved a restricted version of the permutation generation problem, where we generate all the $n!$ permutations of n objects. In this section we consider the general problem of generating all the r-permutations of n objects, where $r \leq n$.

4.4.1 Basic Idea

Suppose we have generated all $n!$ permutations of n distinct objects using Algorithm 3. Now consider suffix of length r of each of these permutations. Each of the r-permutations will appear exactly $(n-r)!$ times. Generalized algorithm is based on the observation that these $(n-r)!$ permutations corresponding to each suffix will be together in the sense that they will correspond to the consecutive numbers (see Theorem 4.8 below). So, in other words the first $(n-r)!$ permutations corresponding to numbers, *i.e*, $0 \ldots (n-r)! - 1$

will have the same suffix and similarly the next $(n-r)!$ permutations will have the same suffix and so on. Thus, we will not generate all these $(n-r)!$ permutations but we will generate only one permutation representative of each suffix.

Observe that the last r numbers in the permutation array depend only on the part $a_{n-r} \ldots a_{n-1}$ of the factorial representation of numbers (see proof of Algorithm 2). This is because, once the swaps corresponding to these numbers are over, then previous suffix of length r does not change, as it does not participate in any more swaps. Hence we can work only with that part of the factorial representation of the numbers and dispense away with the remaining part; this will improve the cost of algorithm from $O(P(n,r) \times n \times \log n)$ to $O(P(n,r) \times r \times \log r)$. However, calculation of $n!$ takes $O(\log n)$ time. So, if the value of $n!$ is provided as part of the input, the algorithm can be made to run with a time of $O(\log r)$ and with the same work as specified above.

Theorem 4.8 *In* Algorithm 3 *all the permutations which have the same suffix will be together (consecutive).*

Proof: In our proof, we use Lemma 1 of Section 3. From Lemma 1, only the prefix of length r of the factorial representation of a number will decide the suffix of length r, of permutation corresponding to that number. Hence unless that prefix changes, our permutation corresponding to the suffix of length r of the original permutation does not change. So we will be done if we are able to show that the prefix of length r changes only after having generated $(n-r)!$ numbers which have the same prefix and that all these numbers are consecutive.

Now from Lemma 1, any natural number k from 0 to $(n-r)!-1$ can be uniquely represented as a sequence $(a_{n-r-1}, a_{n-2}, \ldots, a_1)$ with $0 \leq a_i \leq i$ such that $k = \sum_{i=1}^{n-r-1} a_i i!$

Let $R = (b_{n-1} \ldots b_1)$ is the factorial representation of an arbitrary number s in the range 0 to $n! - 1$. Based on our earlier observations, the prefix $(b_{n-1} \ldots b_{n-r})$ decides the suffix of length r of the s-th n-permutation. Now, let us replace the suffix of length $n-r-1$ of R with $(a_{n-r-1} \ldots a_1)$ which is the factorial representation of an arbitrary number k in the range 0 to $(n-r)!-1$. Then we have got a valid factorial representation. This implies that there are $(n-r)!$ numbers which are consecutive and in which there is no change in their prefix of their factorial representation. This follows from Lemma 1, because, the numbers are unique in this representation. Hence the theorem. ∎

The algorithm is given in Figure 5.

We can clearly see that Algorithm 4 is very similar to Algorithm 3 from which it is derived. Hence we do not conduct here a detailed analysis as the observations made for the original algorithm are applicable to this algorithm *mutatis mutandis*. Here the number of swaps for each permutation is r. This algorithm can also be implemented on a EREW model. Thus we have the following theorems.

Theorem 4.9 Algorithm 4 *has a time complexity of* $O(\log n)$ *and a cost of* $O(P(n,r)r\log r)$ *and runs on a EREW model.* ∎

Theorem 4.10 Algorithm 4 *has a time complexity of* $O(\log n)$ *and a cost of* $O(P(n,r)r\log\log r)$ *and runs on the ARBITRARY CRCW PRAM.* ∎

4.4.2 Generating permutations Lexicographically

Algorithm 1 generated permutations in lexicographic order, but the Algorithms 3 and 4 did not generate permutations in lexicographic order. In this section we suggest a post-processing method based on ranks to get permutations in lexicographic order without any increase in time complexity. We can calculate the rank of the permutation in $O(\log m)$ time with m processors on CREW PRAM (see Section 2.2.1). Using ranking, we can easily show that

Theorem 4.11 *The* m*-permutations of* n *distinct objects can be generated in lexicographic order in* $O(\log n)$ *time with* $O(P(n,m)m\log m)$ *work on the CREW model.*

Theorem 4.12 *The* m*-permutations of* n *distinct objects can be generated in lexicographic order in* $O(\log n)$ *time with* $O(P(n,m)m^2)$ *work on the EREW model.*

4.5 Relationship between generation of permutations and parallel integer sorting

There is a relationship between our algorithm to generate permutations and a restricted version of parallel integer sorting. Consider restricted versions of parallel integer sorting.
Restricted Problem 1 : *Sort* n *integers in given in array* A *where the following property holds.*$A[i] \leq i$ *and the integers are in the range* $1 \ldots n$.

```
Algorithm Gen_perm(n,r)
Begin
  for j = 0 to n!-1 step (n-r)!  in parallel do
1:    for i = n-r to n-1 in parallel do
          a[j,i] = (j mod (i+1)!  )div i!  ;
          b[j,i+1].(key,data) = (i+1 - a[j,i], i+1);
2:    sort b[j] Stably;
      /* Finding minimums */
3:    for i = n-r to n in parallel do
3.1:      c[j,i].min = 0;

3.2:      Assign C[j,i].min = p, in such a way that 'p' is the minimum
          element that took part in a swap with element 'i' for the
          permutation corresponding to integer 'j'.  Use array b[j]

          /* Set the linked list */
3.3:      if c[j,i].min != 0 then
             d[j,i].next = c[j,i].min;
          else
             d[j,i].next = nil;
4:    Compress(d[j]);
5:    for i := n-r to n in parallel do
          if (b[j,i].key != b[j,i+1].key then
             /* b[j,i].data is first index which swaps with index b[i,j].key */
             perm[j,b[j,i].data] = b[j,i].key;
          else
             perm[j,b[j,i].data] = d[j,b[j,i+1].data].next ;
End.
```

Figure 5: Algorithm 4

The next restricted problem removes one part of the above restriction and is formally defined below.

Restricted Problem 2 : *Sort n integers in the range $1 \ldots n$.*

We first show that Restricted Problem 2 is reducible to Restricted Problem 1. More specifically if there exist an $O(\log n)$ time optimal algorithm for the Restricted Problem 1 then there is an algorithm with the same bounds for Restricted Problem 2.

Let us assume that we have to sort n integers in the range $1 \ldots n$ given in an array A. Copy $A[1 \ldots n] + 1$ to $B[n+1 \ldots 2n]$ and assign 1's to $B[1 \ldots n]$. Now in array B all elements are in the range $1 \ldots 2n$ and $B[i] \leq i$ for each i. If there is an $O(\log n)$ optimal algorithm for Restricted Problem 1, then we can sort items of array B using this algorithm; all extra elements will be at one end and can be removed by shifting remaining items n places to left. Later we subtract 1 from the remaining elements to restore the elements. This post processing can be done in constant time. Thus we have got an optimal algorithm for less Restricted Problem 2.

We next show that a relationship exists between parallel integer sorting and our algorithm to generate permutations. Note that in algorithm 2, a permutation is being found for each integer in range 0 to $n!-1$. Hence this can be considered as an *Unranking* procedure.

Theorem 4.13 *If the unranking procedure of* Algorithm 2 *can be implemented in $O(\log n)$ time with $O(n)$ work then we can solve the Restricted Problem 1 of parallel integer sorting in time $O(\log n)$ in an optimal manner.*

Proof: We are generating a permutation corresponding to an integer in a particular range in Algorithm 2. Let us assume it produces a permutation corresponding to a number in time $O(\log n)$ and with a processor complexity of $O(n/\log n)$.

Given a parallel integer sorting problem with the restriction $A[i] \leq i$ we construct an integer which is given to the algorithm for the production of corresponding permutation. We later do post processing to get our sorted list. We first indicate the preprocessing needed.

```
for i =1 to n in parallel do
    B[i-1] = i - A[i];
```

```
    C[i] = i;
prefix_mult(C);
for i =1 to n-1 in parallel do
    D[i] = B[i]*C[i];
P = sum(D);
```

In the above pre-processing we are constructing an integer whose permutation we will later generate. In the generation of permutation, there will be a swap corresponding to each element except the first element. For each i, we intend that there should be a swap of ith element with $A[i]$ th element. This information is encoded in the factorial representation of an integer. So, in the pre-processing step we first obtain the factorial representation (stored in array B) and later obtained the corresponding integer P.

It is easy to observe that the for loop can be implemented in $O(1)$ time with $O(n)$ processors or $O(\log n)$ time with $O(n/\log n)$ processors. After prefix product computation, $C[i] = i!$; all products can be computed in $O(\log n)$ time with $O(n/\log n)$ processors[25].

Now we create another array D as follows: $D[i] = B[i] * C[i]$. We then sum all the elements of array D into P. This gives us the required number. It is easy to observe that pre-processing can be accomplished in time $O(\log n)$ with $O(n/\log n)$ processors. We give number P as input to the permutation generation algorithm.

Let us assume that the permutation generator works optimally and then generates the permutation corresponding this integer in $O(\log n)$ time. We will apply the following post processing to extract the sorted list from the permutation.

```
1 for i = 1 to n in parallel do
      E[i] = 0;
1.1   write i into E[A[i]].
1.2   if E[i] != 0 then
         F[E[i]].next = i;
2 compress(F);
3 for i = 1 to n in parallel do
```

```
// Start of each linked list
if E[i] != 0 then
   Start[i].next = E[i];
if perm[i] == b and A[i] == b then G[i].next = NIL;
else G[i].next = F[perm[i]].next;
```

Here we use a CRCW priority model in which in the case of simultaneous writes the lowest numbered processor succeeds. Here A is the array which has the list to be sorted.

From Step1.1 we can see that, if $E[i] = 'c'$ then 'c' is the smallest element that that has participated with 'i' in swap. In Step 1.2 we form linked lists. In Step 2 we compress these linked lists. In Step 3 we form a set of linked lists for each $i, 0 \leq i \leq n$; these are the elements which are swapped with i or in other words, these are the elements which have the value 'i' as their key. We have a linked list associated with each index i of the list. If we put all these together and rank the linked list[25] them then we obtain the sorted list.

If we analyze the time needed for post processing, we can see that the time needed is essentially $O(\log n)$ with $O(n/\log n)$ processors. As we have already seen the preprocessing also takes the same time and with same processor complexity. Hence the theorem follows. ∎

4.6 Conclusions

Existence of an optimal algorithm for our unranking procedure in permutation generation with time of $O(\log n)$ and $O(n)$ work will result in an optimal algorithm for a restricted case of parallel integer sorting and *vice versa*.

Chapter 5

Making Permutation Generation Optimal

In this chapter we discuss an approach to make a particular class of permutation generation algorithms optimal. For those permutation generation algorithms which do not belong to this class, even though the modified algorithm may not be optimal, it may possibly take less work.

5.1 Algorithm

Let us assume we are given a permutation generation algorithm A, which generates all the r-permutations of n objects with $W(n,r)$ work and $T(n,r)$ time. Using the approach suggested here, we obtain a better algorithm with less cost. The resulting algorithm may also be optimal in certain cases.

Using the algorithm A, we generate all $P(n-1,r-1)$ $r-1$-permutations of $n-1$ objects $\{1,2,\ldots,n-1\}$ in $T(n-1,r-1)$ time with $W(n-1,r-1)$ work. Using these $r-1$-permutations we obtain the r-permutations of n objects $\{1,2,\ldots,n\}$.

We proceed as follows. We assume that n and r values are known to every processor at the start. First the processors find the value of $P(n-1,r-1)$ in $O(\log r)$ time and by making sufficient copies every processor knows the value of $P(n-1,r-1)$. We make n copies of the already obtained $r-1$-permutations on EREW PRAM. A value can be copied n times in $O(\log n)$ time with $O(n)$ work on EREW PRAM[25]. Hence

this copying takes a time of $O(\log n)$ with $O(nP(n-1,r-1))$ work. In r-permutations of n objects, given the first element as i, the suffixes are the $r-1$-permutations of $n-1$ objects $S' = \{1,2,\ldots,n\} - \{i\}$. Each value of i, (for all $1 \leq i \leq n$,) can act as a prefix for r-permutations. So, we attach these prefixes to already obtained $r-1$-permutations to form r-permutations. We need $r-1$-permutations of S' for different values of i. We show below how to obtain the $r-1$-permutations of $n-1$ objects $\{1,2,\ldots,n\}-\{i\}$ for an arbitrary i. Consider each element of the given $r-1$-permutations of $n-1$ objects $\{1,2,\ldots,n-1\}$. If that element is less than i, keep it as it is. If that element is greater than or equal to i then add 1 to it and replace the old value with this new value. We get $r-1$-permutations of S'.

Using this method, we can obtain all the $r-1$-permutations of $n-1$ objects $\{1,2,\ldots,n\}-\{i\}$ for each i. This can be done in constant time with $O(nP(n-1,r-1))$ work on EREW PRAM. Note that there are no concurrent reads as for each i, we work on the ith copy of $r-1$-permutations of $n-1$ objects.

We copy $r-1$-permutations of $n-1$ objects $\{1,2,\ldots,n\}-\{i\}$ into locations $(i-1)P(n-1,r-1)+1$ to $iP(n-1,r-1)$ prefixing each $r-1$-permutation with i. This we do independently and simultaneously for each i in constant time with $O(rnP(n-1,r-1)) = O(rP(n,r))$ work. Hence we have obtained all the r-permutations of n objects $\{1,2,\ldots,n\}$.

We get the following theorem:

Theorem 5.1 *Given an $O(T(n,r))$ time, $O(W(n,r))$ work algorithm for generating r-permutations of n objects, the modified algorithms runs in a time of $O(T(n-1,r-1)+\log n)$ time with $O(W(n-1,r-1)+rP(n,r))$ work on the same model on which the original algorithm runs.*

Clearly, if the original algorithm generates the permutations in lexicographic order, the modified algorithm also generates the permutations in lexicographic order. If our given permutation generation algorithm is such that $W(n-1,r-1)$ is of the order of $O(P(n,r)r)$ then the resultant algorithm based on above theorem will be optimal. The above theorem can be put in a much more broader perspective. If the given permutation generation algorithm is such that its work $W(n-k,r-k)$ is $O(P(n,r)r)$ for some constant k, then also we can make it optimal. The approach is similar. We repeat the

method outlined above k times. That is, we first generate the $r-k$-permutations of $n-k$ objects and using them in k steps, we obtain the r-permutations of n objects, repeating the above method k times.

In our algorithm as given above, we use EREW PRAM. If we use CREW PRAM instead, then we would have avoided the initial copying step (which copies each $r-1$-permutation n times) and we get the following theorem.

Theorem 5.2 *Given an $O(T(n,r))$ time $O(W(n,r))$ work algorithm for generating r-permutations of n objects, on any model at least as strong as CREW, the modified algorithms runs in $O(T(n-1,r-1)+\log r)$ time with $O(W(n-1,r-1)+rP(n,r))$ work on the same model.*

Using algorithm as specified in Chapter 4 (see Theorem 4.12) we get the following result:

Theorem 5.3 *The r-permutations of n objects can be generated lexicographically in $O(\log n)$ time with optimal $O(rP(n,r))$ work on EREW PRAM.*

Chapter 6

Faster Optimal Algorithms for Generation of Derangements

6.1 Introduction

A derangement of set of n integers $\{1, 2, \ldots, n\}$ is defined as a permutation P of these integers which changes every element so that no integer appears in its natural position. ormally, if P is the ordered n-tuple $(p_1 p_2 \ldots p_n)$ then P is a derangement of $\{1, 2, \ldots, n\}$ rovided that $p_i \neq i$ for all i, $1 \leq i \leq n$. The number of derangements of n objects is enoted by $D(n)$. $D(n)$ satisfies the recurrence relation

$$D(n) = (n-1)(D(n-1) + D(n-2)) \tag{6.1}$$

ith $D(0) = 1$ and $D(1) = 0$.

For example, there are nine derangements for $n = 4$, namely

2143 2341 2413 3142 3412 3421 4123 4312 4321

Let $x = (x_1 x_2 \ldots x_n)$ and $y = (y_1 y_2 \ldots y_n)$ be two derangements. We say that x *recedes* y *in lexicographic order* if there exists an integer i, $1 \leq i \leq n$, such that $x_j = y_j$ r all $j < i$ and $x_i < y_i$. Each derangement has an associated index in lexicographic der. By *ranking* we mean getting this index for an arbitrary derangement. *Unranking* the inverse operation of ranking; unranking finds the combinatorial object given its dex in lexicographic order.

6.1.1 History and Related Work

A number of sequential[1, 36] and parallel[23] algorithms exist for generation of derangements. The best known parallel algorithm for generating derangements runs with a time complexity of $O(n \log n)$ with $D(n)$ processors[23]. This algorithm is not cost optimal as generation of derangements has a lower bound of $O(nD(n))$ as we have to generate $D(n)$ derangements each of length n and the sequential algorithm in [1] takes $O(nD(n))$ time.

6.1.2 Preliminaries

We present some of the mathematical concepts that will be used later in the presentation.

The closed form formula for $D(n)$ based on Eq. (6.1) is given below.

$$D(n) = n![1 - \frac{1}{1!} + \frac{1}{2!} - \ldots + \frac{(-1)^n}{n!}] \tag{6.2}$$

It is not difficult to show that [30] $D(n) \approx e^{-1}n!$ where the approximation is quite accurate even for small values of $n = 4$. We can even show that

$$D(n) = \Theta(n!) \tag{6.3}$$

Note that in a derangement $(p_1 p_2 \ldots p_n)$ of n elements, for each position i, $i \neq p_i$, for $1 \leq i \leq n$. That is, every position is restricted. A *partial derangement* of n elements, with k *free* elements is a generalization where k of the elements are without restriction. In other words, k (specific) non-restricted elements can be placed in any of the n positions and each of the remaining $(n-k)$ elements can be placed in any position, other than a restricted position.

We can also define partial derangements as follows. Suppose we have set S consisting of n elements out of which the values of $n-k$ elements are in the range $[1..n]$ and the values of the remaining k elements are not in this range. A partial derangement is a permutation $P = (p_1 p_2 \ldots p_n)$ of these n elements of the set S such that $p_i \neq i$ for $1 \leq i \leq n$. We denote by $D(n,k)$ the number of partial derangements with k free elements.

$D(n,k)$ satisfies the following relation[23]:

$$D(n,k) = D(n,k-1) + D(n-1,k-1) \tag{6.4}$$

Based on this recurrence relation, we can obtain the following expression for $D(n,k)$

$$D(n,k) = \sum_{i=0}^{k} C(k,i)D(n-i) \tag{6.5}$$

where $C(k,i)$ is the binomial coefficient.

6.1.3 Notation

We define our notation which will be used in the remaining part of the chapter. $Rank(D)$ denotes the rank of derangement D. Suppose A and B are derangements, $A \prec B$ iff A lexicographically precedes B. By $|C|$, we mean the cardinality of a set C. $D[i..j]$ denotes the sub-array with indices from i to j. We say k is *in the range* $[i..j]$ iff $i \leq k \leq j$.

6.1.4 Organization of the Chapter

We discuss the $O(n)$ time algorithm in Section 2. We present the ranking function in Section 3. We discuss various implementations of ranking function in Section 4. Section 5 contains the method to obtain derangements given the permutations.

6.2 The $O(n)$ Time Algorithm

Our first algorithm to generate derangements is based on a data structure which we call *Derangement Tree*. A derangement tree is similar to the permutation tree (see Subsection 4.2.1). Like a permutation tree which captures the notion of permutations in the form of a tree a derangement tree does the same for derangements. All the derangements are represented in a succinct and simple manner as the paths in a tree.

6.2.1 Algorithmic Interpretation

We indicate the mathematical basis for the derangement tree which we are going to describe and the algorithm which is based on the concept of derangement tree.

Recall that derangements are related by the recurrence relation given below.

$D(n) = (n-1)(D(n-1) + D(n-2))$

The combinatorial interpretation associated with the derivation of this identity provides a basis through which we have designed the derangement tree.

We are interested in number of derangements with n objects $\{1, 2, \ldots, n\}$ such that we place them in n places. The important restriction is that the i-th position should not contain the i-th object i. We formally express this in a more general form as location i is matched to value i. So, when a location is *matched* to a particular value, the meaning is that location is forbidden to contain the indicated value. Suppose we are interested in generating the derangements of n objects, not necessarily integers in the range $1 \ldots n$, then this notation provides a basis to design our algorithm. We associate each object with a location and generate derangements.

Let us find the number of derangements for n objects – where initially location i is matched to value i. Obviously the position 1 can have $n-1$ possibilities to fill it. More specifically, any i, $i \neq 1$, can come into location 1. Now, let us count for an arbitrary i ($i \neq 1$), the number of derangements with i in first position. Note that 1 can be placed in any position as its restricted location 1 is already filled. We delineate our calculation into two parts. We will first find the number of derangements in which value 1 occupies location i and then find the number of derangements in which value 1 does not occupy location i.

In the first case, since value 1 occupies location i, the position of value 1 is fixed and the remaining $n-2$ locations can be filled in $D(n-2)$ ways.

In the second case, value 1 is forbidden to use location i. Note that since value i is already placed in location 1, location i has got its new matching with value 1 as 1 can not be in location i in this case; In this situation, we have $D(n-1)$ derangements.

As the number of items are $n-1$ for i, the identity follows.

6.2.2 Derangement Tree

We are generating all the derangements of $S = \{1, 2, \ldots, n\}$. We use the tree structure to get the inherent parallelism and reflect the recursiveness in the interpretation.

We assume that $S_1[i] = S_2[i] = i$ for $1 \leq i \leq n$. These arrays are used to represent the fact that location $S_1[i]$ is matched to value $S_2[i]$ for arbitrary i.

We use $remove(S, \{i, j\})$ to denote the array obtained by eliminating the ith and jth indexed elements of S and $append(S, a)$ to denote the extended array obtained by appending the value a to the array S.

Definition : An (n, S_1, S_2)-*Derangement Tree* corresponding to the derangements of n objects present in S_2 with indices specified by S_1 is a tree with the following properties:

1. If $n = 1$ then the root has no children and effectively the tree is empty as there are no derangements.

2. If $n = 2$ then the root has 1 child whose label is $[(S_1[1], S_2[2]), (S_1[2], S_2[1])]$.

3. If $n = 3$ then the root has 2 labeled children such that the label of v_i, the i-th child is $(S_1[1], S_2[i+1])$ and the i-th subtree rooted at v_i is an $(n-1, S_1', S_2')$-Derangement Tree, where $S_1' = append(remove(S_1, \{1, i+1\}), S_1[i+1])$ and $S_2' = append(remove(S_2, \{1, i+1\}), S_2[1])$.

4. If $n > 3$ then the root has $2(n-1)$ labeled children such that the label of v_{2i-1}, the $2i-1$-th child is $(S_1[1], S_2[i+1])$ and the $2i-1$-th subtree rooted at v_{2i-1} is an $(n-1, S_1', S_2')$-Derangement Tree, where $S_1' = append(remove(S_1, \{1, i+1\}), S_1[i+1])$ and $S_2' = append(remove(S_2, \{1, i+1\}), S_2[1])$; the label of v_{2i} is $[(S_1[1], S_2[i+1]), (S_1[i+1), S_2[1])]$ and the $2i$-th subtree rooted at v_{2i} is an $(n-2, S_3', S_4')$-Derangement Tree, where $S_3' = remove(S_1, \{1, i+1\})$ and $S_4' = remove(S_2, \{1, i+1\})$.

It is easy to observe that each subtree of an (n, S_1, S_2)-Derangement tree is a derangement tree. The path (concatenation of labels associated with nodes) from any child of root to a leaf gives the necessary information to form a derangement of n objects. In the derangement tree a combinatorial object is associated with each leaf – the concatenation of the labels associated with the nodes in the path from root to the given leaf. The label (p, q) denotes that the value q should be stored in location p. In other words, each pair specifies the information needed to fill one position of derangement. Note that in some nodes, label contains two pairs, giving the necessary information to fill two positions.

The derangement tree for $n = 4$ is shown in Fig. 1. In the above figure, the arrays S_1, S_2 for each node are specified in a rectangle. The first row specifies the array S_1 and the next row specifies the array S_2. More over, the label associated with each node is shown inside the node. The notation $p : q$ denotes value q should be put in position p. This information has been shown for only few nodes for simplicity.

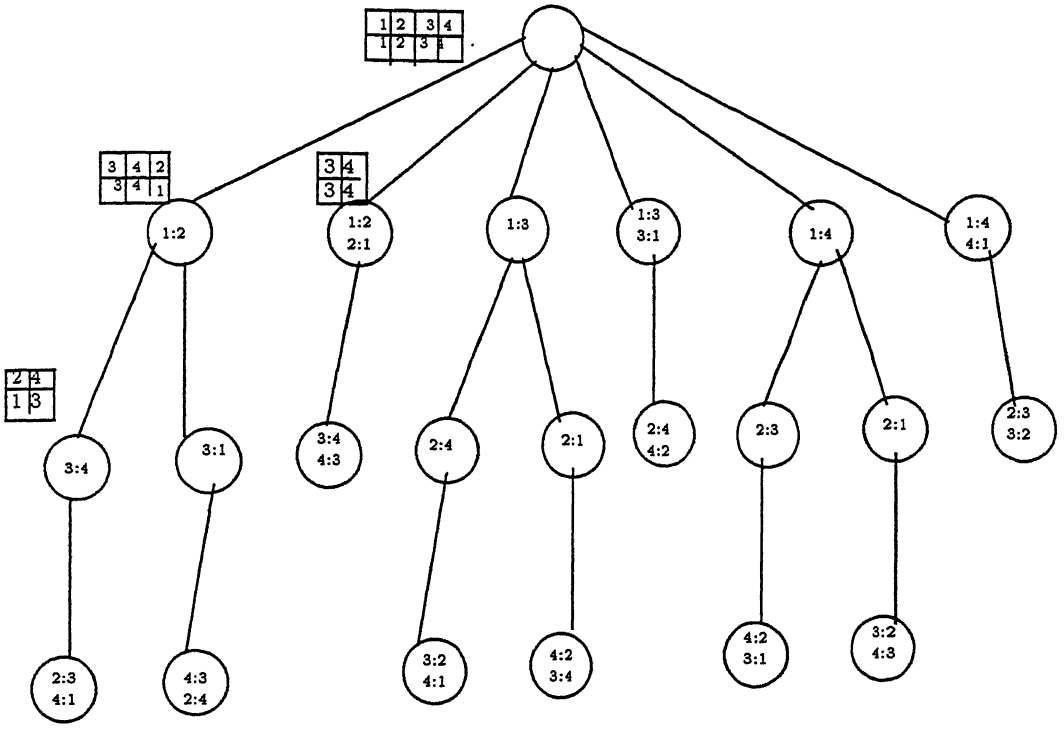


Figure 1: Derangement tree for n =4

6.2.3 Algorithm

Our derangement generation algorithm is based on the above definition of derangement tree. It builds a derangement tree initially and then lists all the root to leaf paths to generate all the derangements. The algorithm is formally shown below.

```
Algorithm Generate_Derangements(n)
Begin
   Calculate the values of D(i) for 0 ≤ i ≤ n
   for i = 1 to n in parallel do
       S1[i] = i ; S2[i] = i
   Create-node(Root);
   Gen_Der(Root, n, S1, S2, 1)
End
Algorithm Gen_Der(Node, n, S1, S2, a)
Begin
```

```
if n = 1 then return NULL
else if n = 2 then
     Create node n;
     n.parent = Node;
     n.label = [(S1[1], S2[2]), (S1[2], S2[1])];
else if n = 3 then
     Create nodes n1, n2;
     for i = 1 to 2 in parallel do
         ni.parent = Node;
         ni.label = (S1[1], S2[i+1])
         si1 = append(remove(S1, {1, i+1}), S1[i+1]);
         si2 = append(remove(S2, {1, i+1}), S2[1]);
         Gen_Der(ni, n-1, si1, si2, a+(i-1)D(n-1));
else
     Create 2(n-1) nodes n[1..2(n-1)];
     for i = 1 to (n-1) in parallel do
         n2i-1.parent = Node;
         n2i-1.label = (S1[1], S2[i+1]);
         s2i-1.1 = append(remove(S1, {1,i+1}), S1[i+1]);
         s2i-1.2 = append(remove(S2, {1,i+1}), S2[1]);

         n2i.parent = Node;
         n2i.label = [(S1[1], S2[i+1]), (S1[i+1], S2[1])];
         s2i.1 = remove(S1, {1,i+1})
         s2i.2 = remove(S2, {1,i+1})
     for i = 1 to 2(n-1) in parallel do
         if i is even then
            Gen_Der(ni, n-2, si.1, si.2, a + (i-2)/2 (D(n-1) + D(n-2)) + D(n-1))
         else
             Gen_Der(ni, n-1, si.1, si.2, a + ⌊i/2⌋.(D(n-1) + D(n-2)))
if n ≤ 2 then
```

```
        A processor is associated with each leaf and it stores
        the derangement associated with it in the index associated with it.
End
```

We are interested in generating the derangements of elements from $S = \{1, 2, \ldots, n\}$. The algorithm is recursive. The base cases when $n = 1, 2, 3$ are explained in the algorithm and are straight forward.

Now, we consider the case when $n > 3$. The root has $2(n-1)$ children and out of which $n-1$ children corresponds to recursive calls for derangements of $n-1$ objects and the remaining for derangements of $n-2$ objects.

We allocate $n-2$ processors with each of the child corresponding to derangements of $n-1$ objects (the odd numbered nodes) and we allocate $n-3$ processors with even nodes. First in each case, one among the processors allocated for each child copies the label from S_1, S_2 as specified in the definition. In the next two steps, all the processors associated with each child form S_1, S_2 to execute the recursive call associated with each child.

The last but one step in the above algorithm is a parallel recursive call. The fifth parameter in the recursive call is specifically provided for taking care of the processor allocation. We try to identify what are the processors that are allocated to each procedure call. The above parallel recursion can be implemented by a model proposed in [9].

Let us consider the procedure call $Gen\_Der(Node, n, S_1, S_2, a)$; here we are interested in generating the derangements of n objects where the matching is specified by the arrays S_1, S_2. More over the processors a to $a + D(n) - 1$ should be allocated for this procedure call . The last parameter specifies that the leafs for this procedure call will store their associated derangements in the locations a to $a + D(n) - 1$.

The algorithm is building the tree level by level. Building a single level takes constant time. In other words, we have the following theorem.

Theorem 6.1 *All $D(n)$ derangements can be generated in $O(n)$ time with $O(D(n))$ processors on the CREW PRAM.*

Proof: We will prove the theorem by mathematical induction on n. We first prove that the time complexity is $O(n)$.

When $n = 2$, the algorithm takes only one step. If $n = 3$ then the algorithm takes two steps. Let us assume the algorithm for $n \leq k$ takes at most cn time for some constant c.

Now, we consider the case when $n = k + 1$. From the algorithm, we can see that after taking a constant number of steps, there are $2(n-1)$ recursive calls. All of these execute simultaneously. Note that half of the recursive calls correspond to the case where $n = k - 1$ and the other half to $n = k$. Hence the time complexity is, by using induction hypothesis, $T(k) + O(1) \leq c_1 + max(ck, c(k-1)) = c_1 + ck \leq c(k+1) = cn$, if $c_1 \leq c$. Hence our first claim is validated.

We next show again by induction that the number of processors needed are $D(n)$. For $n = 2$, or 3, this is obvious. Let us assume the claim is valid for $n \leq k$.

For the case where $n = k + 1$, there are $2(n-1)$ recursive calls to *Gen_Der* and by induction hypothesis, i-th recursive call takes $D(n-1)$ processors ($D(n-2)$ processors) if i is odd (even). Hence the number of processors needed for the recursive calls are $(n-1)(D(n-1) + D(n-2) = D(n)$.

Further, we need $n-2$ $(n-3)$ processors for the i-th call if i is odd (even). So, the number of processors needed, by induction hypothesis, are $max((n-1)(D(n-1) + D(n-2), (n-1)(n-2+n-3))$.

Since, $D(n) \geq n - 1$ for $n \geq 2$, it follows that, we need $D(n)$ processors for our recursive call, verifying our other claim also.

Note that in our algorithm, there are simultaneous reads (reading the arrays S_1,S_2, for example) but no simultaneous writes and so we can implement the algorithm on CREW PRAM. ∎

Recall that derangement generation has a lower bound of $O(nD(n))$. Hence our algorithm is cost optimal.

6.3 Ranking

Let $D = (d_1 d_2 \ldots d_n)$ be a derangement of n objects. The problem of ranking D is to find the index of D among all the derangements of n objects in lexicographic order.

We proceed as follows. We find the total number of derangements which lexicographically precedes D. Suppose it is r. Then $Rank(D)$ is $r + 1$. We denote by C, the

set of all derangements which lexicographically precede D. So, obviously

$$Rank(D) = |C| + 1 \tag{6.6}$$

We express C as a union of some smaller sets. Suppose a derangement $A = (a_1 a_2 \ldots a_n) \prec D$ then there exists an i such that $1 \leq i \leq n$ and $a_j = d_j$ for $j < i$ and $a_i < d_i$. Note that, the existence of the i in the above definition is *unique*. There are n possible values for i varying from 1 to n. We put together all the derangements which lexicographically precede D for the same i in a set denoted $C(i)$. So, we will have n sets. Formally,

Definition: $C(i) = \{A = (a_1 a_2 \ldots a_n) | A$ is a derangement, $a_j = d_j \forall j < i$ and $a_i < d_i$ $\}$

It is not difficult to prove the following lemmas based on the above definition.

Lemma 6.2 $C = \bigcup_{i=1}^{i=n} C(i)$

Lemma 6.3 $C(i) \cap C(j) = \emptyset$ *if* $i \neq j$

From the above two lemmas and Eq. (6.6), the following theorem follows

Theorem 6.4

$$Rank(D) = 1 + \sum_{i=1}^{n} |C(i)| \tag{6.7}$$

So, our aim is to find the value of the expression $\sum_{i=1}^{n} |C(i)|$. We try to find the value $|C(i)|$ for an arbitrary i, $1 \leq i \leq n$. The calculation can be repeated for all applicable values of i to get the respective values of $|C(i)|$.

By finding the value of $|C(i)|$, we mean finding the number of derangements $A = (a_1 a_2 \ldots a_n)$ which have the prefix $(d_1 d_2 \ldots d_{i-1})$ but $a_i < d_i$. Note that the values we can use as a_i will be coming from the set of elements contained in $D[i+1..n]$, i.e, $S = \{d_{i+1}, d_{i+2}, \ldots, d_n\}$. Suppose we have selected an element $a_i \in S$ such that $a_i < d_i$. Thus, we have fixed first i positions of A. We have to fill positions $i+1$ to n. Obviously these positions will be filled with elements of the set $S' = S - \{a_i\} + \{d_i\}$. Let us denote by v_1 the number of elements in the set S' which are *not* in range $[i+1..n]$. The number of ways such that we can put the elements of set S' in the positions $i+1$

to n of A such that $a_j \neq j$, where $i+1 \leq j \leq n$ is $D(n-i, v_1)$ – the number of partial derangements of $n-i$ elements with v_1 free elements. We can repeat the above procedure for each a_i of S. Instead, we classify those elements from S that can come as a_i into two groups. The detailed algorithm is given below.

The ranking function has the following steps.

We denote by S the set of elements in $D[i+1..n]$.

1. Calculate k_1, the number of x_i's, $x_i \in S$ such that $x_i < d_i$ and $x_i \neq i$ and it is in range $[i+1..n]$.

2. Calculate k_2, the number of x_i's, $x_i \in S$ such that $x_i < d_i$ and $x_i \neq i$ and *not* in range $[i+1..n]$.

3. Calculate k_3, the number of x_i's, $x_i \in S$ such that x_i is *not* in range $[i+1..n]$.

Using the values of k_1, k_2 and k_3, we obtain the following expression for $|C(i)|$:

$$|C(i)| = k_1.D(n-i, k_3 + [not((i+1) \leq d_i \leq n)]) + k_2.D(n-i, k_3 - [i+1 \leq d_i \leq n]) \tag{6.8}$$

where, the Iverson's notation $[P]$ is defined as 1 if the predicate P is true and 0 otherwise.

So, we can use the above general procedure for each i, $1 \leq i \leq n$, to find the values of $|C(i)|$ and use Eq. (6.7) to obtain the rank of derangement D.

Our ranking function is designed in such a way that it is more *general* than just needed for ranking a derangement. More precisely, it can be directly used for ranking any partial derangement. Though we are not exploiting this capability of our ranking function in the present chapter, it is expected that such a general nature of ranking can have applications in other situations.

6.4 Implementation of Ranking function

The ranking function given above works quite well if we intend to implement the ranking on a serial computer. But our intention is to implement the ranking in parallel. Note that in a parallel model, we assume in what follows that the value n is available to each

processor at the start. We show the implementation of ranking on EREW and CREW PRAMs. Both of them assume that the pre-processing given below has taken place before their invocation. We first describe the pre-processing steps.

6.4.1 Pre-Processing

The pre-processing involves calculation of (a) $i!$, (b) $D(i)$ for $0 \leq i \leq n$ and (c) $C(i,j)$, (d) $D(i,j)$ for $0 \leq i,j \leq n$.

To calculate $i!$ for $0 \leq i \leq n$, we fill the array E such that $E[i] = i$ (except $E[0] = 1$) and apply prefix multiplication on it. Then, $E[i] = i!$ for $0 \leq i \leq n$. Then, we copy each $i!$, n times so that there will not be any read conflict while calculating binomial coefficients. This can be done in $O(\log n)$ time with $O(n^2)$ work.

Note that a single value can be copied to p locations in $O(\log p)$ time with $O(p)$ work on EREW PRAM[25]. Similarly, prefix sum or prefix multiplication or just sum of n values can be accomplished in $O(\log n)$ time [25] with $O(n)$ work on EREW PRAM.

We use the following well known formula for finding the value of binomial coefficients $C(i,j)$ for $0 \leq i,j \leq n$ and can be done in constant time with n^2 processors.

$$C(i,j) = \frac{i!}{(i-j)!j!} \tag{6.9}$$

with $C(i,j) = 0$ if $i < j$.

After this, we copy each $C(i,j)$, n times to avoid read conflict while finding $D(i,j)$'s. The copying can be done in $O(\log n)$ time with $O(n^3)$ work.

Then we use Eq. (6.2) to find the values of $D(i)$, for $0 \leq i \leq n$. This can be done in $O(\log n)$ time with n processors. The main task involved is a prefix summation. We copy each $D(i)$, n^2 times to avoid read conflict while finding $D(i,j)$'s, which takes $O(\log n)$ time with $O(n^3)$ work.

Later, we use Eq. (6.5) to calculate $D(i,j)$, for $0 \leq i,j \leq n$, in $O(\log n)$ time with $O(n^3)$ work.

Lemma 6.5 *Pre-processing can be done in $O(\log n)$ time with $O(n^3)$ work on EREW PRAM.*

6.4.2 The EREW Implementation

We can find the values of k_1, k_2 and k_3 for each $|C(i)|$ as specified in Algorithm (see Section 6.3).

We would like to rank the derangement $D = (d_1 d_2 \ldots d_n)$. We find the values of $|C(i)|$ for each i, $1 \leq i \leq n$. We will calculate all these independently and simultaneously in parallel. But to calculate $|C(i)|$, we will access the locations from i to n of D. On a CREW PRAM this creates no problem, but a read conflict will arise in the case of EREW PRAM.

To avoid the read conflict, we ensure that each calculation of $|C(i)|$ proceeds on its own copy of D. We get n copies of D by copying each d_i, n times in $O(\log n)$ time [25] with $O(n^2)$ work.

The calculation of $|C(i)|$, we describe below should be repeated simultaneously and independently for each i, $1 \leq i \leq n$ on their own copy of D.

First the processors allocated for the calculation of $|C(i)|$ broadcast the value d_i, n times so that read conflicts are avoided in the calculation of k_1, k_2 and k_3 (see Section 6.3). We describe below a general approach to calculate any of k_1, k_2 and k_3.

Assign n processors to the i-th copy of array D such that j-th processor is assigned to location j of array D. The processors assigned to locations $D[1..i-1]$ write 0 into an array L in their corresponding locations. Other processors look at the value contained in their location of array D and test a boolean condition (this varies depending upon whether we are finding k_1, k_2 or k_3) and write 1 or 0 into corresponding location of array L. Then we sum all the values in the array L in $O(\log n)$ time with $O(n)$ work which gives the value of k_1, k_2 or k_3 depending upon the boolean condition used.

Once we obtained k_1, k_2 and k_3, we use Eq. (6.8) to calculate the value of $|C(i)|$.

After we have obtained the value of $|C(i)|$ for each i, we use Eq. (6.7) to get the value of $Rank(D)$ in $O(\log n)$ time with $O(n)$ work.

Theorem 6.6 *The rank of a derangement can be found in time $O(\log n)$ with $O(n^2)$ work on EREW PRAM if the pre-processing is already assumed to have taken place.*

On the other hand, if we do not consider the pre-processing separately then,

Theorem 6.7 *The rank of a derangement can be found in time $O(\log n)$ with $O(n^3)$ work on EREW PRAM.*

6.4.3 The CREW Implementation

After pre-processing has been done, the calculation of k_1, k_2 and k_3 for each $|C(i)|$ remains (see Section 6.3). Unlike the EREW model in which we independently worked for the calculation of each $|C(i)|$, the approach here is different. We will form arrays K_1, K_2, K_3 such that $K_1[i]$ gives the value of k_1 for $|C(i)|$ and so on. Recall that we are ranking the derangement $D = (d_1 d_2 \ldots d_n)$. Here $D[i]$ is d_i. We proceed as follows for forming the array K_3.

For each i, we initially calculate how many elements of $D[i..n]$ are *in* the range $[i..n]$ as $M[i]$. Then, the number of elements which are *not* in this range, i.e, $K_3[i-1] = n - i + 1 - M[i]$.

Formally, (we depend upon the indentation to show the boundaries of statements.),

Assume that arrays M_1 and M_2 are initialized with zeroes.

```
for i = 1 to n in parallel do
    if i ≤ D[i] ≤ n then M1[i] = 1
    else M2[D[i]] = 1;
    M[i] = M1[i] + M2[i];
Suffix-Sum(M);
for i = 2 to n in parallel do K3[i − 1] = n − i + 1 − M[i];
```

In the above algorithm, for arbitrary j, if $j \leq D[j] \leq n$, then $D[j]$ contributes 1 to each of the $M[t]$ for $1 \leq t \leq j$. Otherwise, $D[j]$ contributes 1 to each of $M[t]$ for $1 \leq t \leq D[j]$.

Clearly algorithm correctly finds $K_3[i]$ for each i.

We next look at the problem of forming of arrays K_1 and K_2. For the calculation of k_1's and k_2's we use *Generalized Prefix Computation* (see Section 2.1) Generalized prefix computation can be done in $O(\log n)$ time with n processors on CREW PRAM[39]. Before proceeding further, we would like to know where each element is occurring in D.

for $i = 1$ to n in parallel do $W[D[i]] = i$;

The integer i is occurring at position $W[i]$ of D.

Returning to the computation for finding of K_1 and K_2, we copy $D[i]$ to $y[n-i+1]$ for each i in a single step and fill the array f with 1's. The '$*$' is the usual summation operator '$+$' and '$<$' is the usual less-than symbol '$<$'. Apply Generalized prefix computation on this, resulting in array E. We copy $E[i]$ to $E_1[n-i+1]$ for each i. For arbitrary i, $E_1[i]$ is the number of elements in $D[i+1..n]$ which are less than $D[i]$.

Now, we form the array K_{12} such that $K_{12}[i] = k_1 + k_2$ of $|C(i)|$ i.e, $K_{12}[i]$ represents the number of elements in $D[i+1..n]$ which are less than d_i and and not equal to i.

If $(i+1 \leq W[i] \leq n)$ and $(d_i > i)$ then $K_{12}[i] = E_1[i] - 1$ else $K_{12}[i] = E_1[i]$

Using the arrays E_1, M and K_{12}, we obtain the arrays K_1 and K_2.

If $i+1 \leq D[i] \leq n$ then

$n - i - E_1[i]$ values of $D[i+1..n]$ are in the range $[i+1..n]$ and greater than d_i.

$M[i+1]$ values of $D[i+1..n]$ are in the range $[i+1..n]$.

Thus, $M[i+1] - (n - i - E_1[i])$ elements of $D[i+1..n]$ are in the range $[i+1..n]$ and less than d_i.

$K_1[i] = M[i+1] - (n - i - E_1[i])$;

$K_2[i] = K_{12}[i] - K_1[i]$;

else

$K_1[i] = 0$; $K_2[i] = K_{12}[i]$

Using K_1, K_2 and K_3 and Eq. (6.8) we get $|C(i)|$ for each i and then use Eq. (6.7) to get $Rank(D)$.

Theorem 6.8 *We can find the rank of a derangement of n objects in $O(\log n)$ time with $O(n \log n)$ work on CREW PRAM when pre-processing is already assumed to have taken place.*

We have divided the calculation of rank into two parts – pre-processing and main processing.

We can imagine a situation where we are given a set of derangements of n objects and we have to rank all of them. In such a situation, it will be better if we do those calculations which are same for each derangement –calculations done in pre-processing –

only once instead of doing them for each and every derangement. Suppose, we have to rank z number of derangements of n objects. Then the work is $(z * n \log n + n^3)$ when the pre-processing is done only once. On the other hand, if we repeat the pre-processing for each derangement, then the work done is $O(z * n^3)$. If $z = \omega(1)$ then our first approach is definitely better than the second one and hence proves the importance of distinct pre-processing.

6.5 Derangements from Permutations

Every derangement is a permutation (throughout this section, by permutation we mean n-permutation) but not *vice versa*. So, if we are given all the permutations of n objects, they obviously contain all the derangements of n objects. Let us assume we are given all the permutations of n objects in lexicographic order. Then the problem is to put together all the derangements among them in lexicographic order. The following lemma is trivial:

Lemma 6.9 *The ordered list of derangements among all the permutations of n objects in lexicographic order will also be in lexicographic order. More over this list contains all the derangements of n objects.*

Given the lemma, the easy approach to generate derangements from permutations in parallel is like this. First we identify those permutations which are derangements and later *rank* each of them in parallel and store them in the location given by their rank. But the problem with this approach is it has very high cost. If we use our $O(n^3)$ work and $O(\log n)$ time ranking algorithm (see Theorem 6.7) we will be able to generate the derangements in $O(\log n)$ time with $O(n^3 n!)$ work which is clearly not optimal. So, now the question is how to generate the derangements optimally from the permutations in lexicographic order with the same time as above. We propose a novel method to tackle this problem.

Instead of ranking *all* derangements, we rank only a *selected set* of derangements and the remaining derangements will obtain their ranks from the ranks of the selected set. More precisely, we divide all the $n!$ permutations into sets of n^2 consecutive permutations (the last set may contain less than n^2 permutations) each. Now we rank only the *first* derangement, called *leader*, from each set (if any) and then the remaining derangements

of the set will get their ranks by prefix sums using the rank of their leader. This approach works because the permutations that are given to us are in lexicographic order.

6.5.1 Description of Algorithm

We divide the given $n!$ permutations into $\lceil \frac{n!}{n^2} \rceil$ sets. Except possibly the last set all the other sets contain n^2 elements each. More precisely, the i-th set contains the permutations with ranks from $(i-1)n^2+1$ to in^2. The processing we are going to do for each set is same. We thus indicate the processing for an arbitrary set, say the i^{th} set, and this processing should be done simultaneously and independently for each set.

The i^{th} set contains the n^2 permutations indexed from $a_1 = (i-1)n^2+1$ to $b_1 = in^2$. For all the n^2 permutations, we would like to determine which of them are derangements. In a derangement the j th element should not be j for $1 \leq j \leq n$. So, in $O(\log n)$ time and $O(n)$ work for each permutation we can determine whether it is a derangement or not on EREW PRAM. The task mainly involves finding the OR of n bits for each permutation. In $O(\log n)$ time with $O(n^3)$ work, we will classify each permutation of our set as a derangement or not. Based on this, we will form an array H such that $H[j] = 1$, if the j-th permutation is derangement or $H[j] = 0$, otherwise.

The next task is to find the *leader* – the derangement whose rank we intend to find explicitly. We will define the leader for a set to be the first derangement in that set. Rephrasing, we need to find the position of first 1 in the array H. The finding of the first 1 can be done by a single application of prefix sum[25] on H. Let us denote by q the index of first 1 in H. Note that it is possible that the index q may not exist. This situation occurs when our set does not contain any derangements. In such a case the processors which are assigned to this set remain idle through out the processing. In any case, the leader of a set, if exists, can be found in $O(\log n)$ time with $O(n^2)$ work.

We rank this leader derangement using our algorithm for ranking on EREW model in $O(\log n)$ time with $O(n^3)$ work (see Theorem 6.7). Note that we are using the version of ranking where pre-processing is part of ranking. Let us denote by s the rank of the leader. We set $H[q] = s$.

Apply prefix sum on H to get the array P. After this step, every derangement of the i-th set has got its rank. Suppose, for arbitrary j, $H[j] = 1$, then the rank of

j-th permutation, which is a derangement is $P[j]$. This follows because the ordered derangements in the i-th set are consecutive derangements in lexicographic order.

Once every derangement of the set has got its rank, we assign n processors to each permutation and those processors which are assigned to derangements write the derangement in the location indexed by its rank in an array meant for storing all the derangements of n objects. Those processors which are assigned to non-derangements remain idle.

Summarizing, the derangements in a single set can be ranked in time $O(\log n)$ with $O(n^3)$ work on EREW PRAM.

As we have $\lceil \frac{n!}{n^2} \rceil$ sets, the work for the entire algorithm is $O(n!n)$. It follows from Eq. (6.3) that,

Lemma 6.10 *The derangements in lexicographic order can be obtained from the permutations of n objects in lexicographic order in $O(\log n)$ time with optimal work $O(nD(n))$ on EREW PRAM.*

The theorem given below follows from the above lemma as permutation generation has a lower bound of $O(n! \times n)$ and since EREW is the weakest among PRAM models.

Theorem 6.11 *Given an $O(T)$ time and $O(W)$ work algorithm for lexicographic generation of all permutations of n objects, we can generate all the derangements lexicographically in $O(T + \log n)$ time with $O(W)$ work on the same model on which permutation generation algorithm runs.*

In Chapter 5 (see Theorem 5.3) we have described an $O(\log n)$ time and $O(n!n)$ work algorithm for lexicographic generation of permutations on EREW PRAM. Using this algorithm and above theorem we obtain the following result.

Theorem 6.12 *All the derangements of n objects can be generated in lexicographic order in $O(\log n)$ time with $O(nD(n))$ work on CREW PRAM.*

Chapter 7

Faster Optimal Parallel Algorithms for the Generation of Combinations

7.1 Introduction

In this chapter we discuss new parallel algorithms for enumerating combinations. We begin with some definitions. Let S be a set consisting of n distinct items say, the first n positive integers, i.e, $S = \{1, 2, \ldots, n\}$. An r-combination of S is obtained by selecting r distinct integers out of S and arranging them in *increasing* order. Thus for $n = 6$ and $r = 3$, one 3-combination is (2 4 6). Two r-combinations are *distinct* if they differ with respect to the items they contain. The number of distinct m-combinations of n items is denoted by $C(n, m)$, where $C(n, m) = \frac{n!}{(n-m)!m!}$.

Thus for $n = 4$, there are four distinct 3-combinations. In the special case where $r = n$, $C(n, n) = 1$.

Now, let $x = (x_1 x_2 \ldots x_n)$ and $y = (y_1 y_2 \ldots y_n)$ be two r-combinations of S. We say that x *precedes* y in *lexicographic order* if there exists an integer $i, 1 \leq i \leq r$, such that $x_j = y_j$ for all $j < i$ and $x_i < y_i$. If a combination A precedes lexicographically another combination B, we denote it by $A \prec B$.

Each r-combination has an associated index in lexicographic order . By *ranking* we mean getting this index for an arbitrary r-combination. Similarly, *unranking* means getting the combination given its index. So unranking is the inverse operation of ranking.

Our algorithm in Section 2 runs with a time complexity of $O(r)$. In Section 3 and the

Section 4 we will see two algorithms which are much faster. There is a thread running between both these algorithms; both are designed based on induction and recursion. They are inspired by [32] in which induction and recursion are used as a design technique to derive new and efficient algorithms.

In the case of generation of r-combinations of n objects there are two parameters that are involved *viz.* r and n. The recursion can be applied on either of these parameters. The algorithm in Section 3 correspond to recursion on the parameter r (r-recursive algorithm) while that of Section 4 correspond to the parameter n (n-recursive algorithm).

7.1.1 History and Related Work

The problem of generation of combinations has a number of sequential[28, 36, 43] and parallel algorithms for its solution. An algorithm for generating all combinations in time $O(C(n,r)/N)r\log r)$ time with N processors appeared in [21]. A cost optimal parallel algorithm using r processors was presented in [10]. The algorithms in [2, 4] can employ up to $C(n,r)/n$ processors.

Previous cost-optimal algorithms for generating combinations have all used the following strategy: a few combinations are generated by unranking, and the remaining are generated by an optimal algorithm for generating the next combination[2]. The algorithms in [4, 10, 41] parallelized the computation of the next combination to take constant time. The algorithms in [4, 41] run on a linear array of processors. The unranking algorithm used is a sequential algorithm which required $O(r^2)$ time per combination. The algorithm in [34] performs the unranking computation optimally and in parallel. It runs with a time complexity of $O(r)$ with optimal number of processors. Our algorithms are different and are based on a divide-and-conquer principle.

7.1.2 Preliminaries

In this section, we present few basic concepts related to combinations for later easy reference. The reader is referred to [26] for proofs.

$$C(n,r) = \frac{n!}{(n-r)!r!} \tag{7.1}$$

$$C(n,r) = C(n,n-r) \tag{7.2}$$

$$C(n,r) = 0,\ when\ n < r \tag{7.3}$$

$$C(n,r) = \frac{(n)(n-1)(n-2)\ldots(n-r+1)}{(r)(r-1)(r-2)\ldots(1)} \tag{7.4}$$

$$C(n,r_1) \leq C(n,r_2), if\ 0 \leq r_1 \leq r_2 \leq \lceil \frac{n}{2} \rceil \tag{7.5}$$

$$C(n,r_1) \geq C(n,r_2), if\ \lfloor \frac{n}{2} \rfloor \leq r_1 \leq r_2 \tag{7.6}$$

$$C(n_1,r) \leq C(n_2,r), if\ n_1 \leq n_2 \tag{7.7}$$

7.2 $O(r)$ Time Algorithm

Our first algorithm to generate combinations is based on a data structure which we call *Combination Tree*. A combination tree is similar to the permutation tree (see Section 4.2.1). Like a permutation tree which captures the notion of permutations in the form of a tree a combination tree does the same for combinations. All the combinations are represented in a succinct and simple manner as the paths in a tree.

7.2.1 Algorithmic Interpretation

We indicate the mathematical basis for the forthcoming algorithm which is based on building a tree of combinations.

Consider the well known combinatorial identity[20]:

$$\sum_{k=0}^{n-1} C(k,r) = C(0,r-1) + C(1,r-1) + \ldots + C(n-1,r-1) = C(n,r) \tag{7.8}$$

The above identity can be given the following algorithmic interpretation when we are generating the r-combinations for n objects $\{1 \ldots n\}$. It is based on the concept that in

a combination the values are in increasing order. Suppose we consider r-combinations with prefix "1".The Eq. (7.8) says that there are $C(n-1,r-1)$ r-combinations with this prefix. More over the suffixes are nothing but $r-1$-combinations of $n-1$ objects $\{2,\ldots,n\}$. In general, with "i" as the prefix, there are $C(n-i,r-1)$ r-combinations where the suffixes are the $(r-1)$-combinations of $n-i$ objects $\{i+1\ldots n\}$. Obviously the prefix can be "i" where $1 \leq i \leq n$. The above identity indicates all these possibilities. But note that if $n-i < r-1$ then there will not be any r-combinations with prefix "i" because then $C(n-i,r-1) = 0$. The definition of combination tree, given below, uses this idea.

7.2.2 Combination Tree

We are generating all r-combinations of n elements of S. We assume that the elements of S can be ordered. Let $S[i]$ denote the ith element of set S. Similarly, we assume that $S[i\ldots j]$ denote array consisting of all the elements from $S[i]$ to $S[j]$.

Definition: An (n,r,S)-Combination Tree corresponding to the r-combinations of n objects present in S is a tree with the following properties:

1. if $r = 1$ then the root has n children and ith child is labeled with the $n-i+1$th element of S.

2. if $r > 1$ then the root has $n-r+1$ labeled children such that the label of v_i, the ith child, is $S[n-r+2-i]$ and the ith subtree rooted at v_i is an $(r+i-2, r-1, S[n-r+3-i\ldots n])$-Combination tree.

It is easy to observe that, each subtree of an (n,r,S)-Combination tree is a combination tree. The path (the concatenation of labels associated with the nodes in order) from any child of root to a leaf is an r-combination of n objects and we call this the *combination associated* with that leaf. Combinations associated with different leaves are distinct. Furthermore, the combination tree preserves the lexicographic order of the combinations in the sense that if a leaf v is to the *right* (defined below) of another leaf w, then the combination corresponding to v lexicographically precedes that of w. A leaf p is *left* of a leaf q if and only if there exists nodes s,t,u such that p is a descendant of t and q is a descendant of u and t and u are children of s and t occurs before u (for

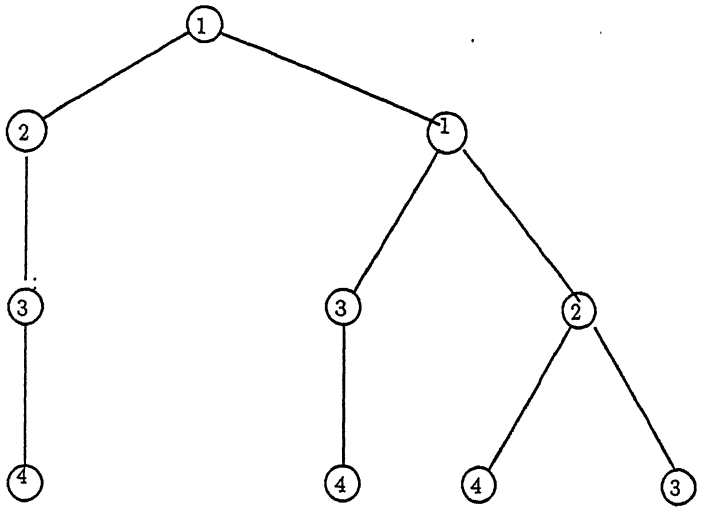


Figure 1: Combination Tree when n = 4 and r = 3

the relevant graph theory terminology refer to [14]). We give an example of combination tree in Fig.1.

7.2.3 Algorithm

Our algorithm builds a combination tree initially and then lists all the root to leaf paths to generate the required combinations. The algorithm works as follows.

We are interested in generating the r-combinations of elements from $S = \{1, 2, \ldots, n\}$. The root has $n - r + 1$ children and we associate a processor with each child. This processor copies its corresponding label and maintains the information needed so that a recursive call can be made. This processor also maintains the parent information needed to maintain the tree structure.

Once the tree is built, we associate a processor with each leaf and it reads the path from root to itself to get the corresponding combination. A number is provided with each processor which is helpful in storing the combination so that we can list the combinations in lexicographic order.

The algorithm is formally given in Fig. 2. In the algorithm, we depend upon the indentation to show the boundaries of statements.

We explain the salient points of the algorithm. A label is associated with each node and it serves dual purpose. First, it represents the corresponding value in a combination.

Second, the label is useful in identifying the set of objects for which we are generating combinations for a particular recursive call. For example, if the label of a node is 'k' then it means that this node is associated with a recursive call in which we need to generate the p-combinations for $k+1$ to n objects; here $r-p$ is the length of the path from root to the given node.

The last step in the above algorithm is a parallel recursive call. One parameter in this recursive call is specifically provided for taking care of the processor allocation. It is a convention in general to identify which processors should be allocated for each procedure call. This information in general is provided through passing parameters to the procedure call. In fact, we can allocate the processors based on the model proposed by [9] using generic processor allocation (see Section 2.3 and Eq. (7.8)).

Let us consider the procedure call $Gen\_Comb(node, n, r, a)$. This procedure call says that we are interested in generating r-combinations of n objects. More over, the processors a to $a + C(n,r) - 1$ should be allocated for this procedure call. In the above procedure call this processor allocation is facilitated by Eq. (7.8).

7.2.4 Proof of Correctness

Theorem 7.1 *Algorithm correctly generates all the r-combinations in lexicographic order.*

Proof: We prove a much stronger statement. We prove that $Gen\_Comb(node, n, r, a)$ with $node.label = x$, generates the r-combinations of n objects $\{x+1, x+2, \ldots, x+n\}$ and stores them in locations a to $a + C(n,r) - 1$ such that they are in decreasing lexicographic ordering (ordinary lexicographic order is increasing). The theorem is a special case when $nodel.label = 0$ and $a = 1$.

We prove this statement by induction on r. When $r = 1$, the statement is clearly true. Assume that the statement holds for $r-1$ and we prove that it works for r.

Consider the call, $Gen\_Comb(node, n, r, a)$. There are $C(n-i, r-1)$ combinations with $x+i$ as the first value in the r-combination. In decreasing lexicographic order, we have first the $C(r-1, r-1)$ r-combinations with $x+n-r+1$ as the first value in decreasing lexicographic order and followed by $C(r, r-1)$ r-combinations with $x+n-r$ as the first value in decreasing lexicographic order and so on. In general, with $x+n-r-i+2$

```
Algorithm Generate_Combinations(n,r)
Begin
   N = n; R = r;
   Create_node(Root);
   Root.label = 0;
   Gen_Comb(Root,n,r,1);
End;

Algorithm Gen_Comb(node, n, r, a)
Begin
   Create n-r+1 nodes, Node[1..  n-r+1];
   /* Copy labels */
   for i = a to a+n-r in parallel do
       Node[i-a+1].parent = node;
       Node[i-a+1].label = n - r + 1 + node.label - (i - a) ;
       if (r == 1) then
          Write the reversal of the path from Node[i] to the child
          of the root as the combination in location i.
          // Write in location C(N, R) - i+1 for lexicographic order.
   for i = 1 to n-r+1 in parallel do
       Gen_Comb(Node[i], r-2+i, r-1, a + C(r-2+i, r));
End.
```

Figure 2: Algorithm to build a Combination Tree

as the first value, $\sum_{j=1}^{i-1} C(r-2j, r-1) = C(r-2+i, r)$ combinations precede it. So, the combinations with $x+n-r-i+2$ as first value should start at $a+C(r-2+i,r)$ in decreasing lexicographic order. More over there are $C(n-(n-r-i+2), r-1) = C(r+i-1, r-1)$ combinations with $x+n-r-i+2$ as first value. By mapping that i^{th} node in our algorithm to $x+n-r-i+2$ as the first value, our claim is true by induction hypothesis. ∎

7.2.5 Analysis

The algorithm is building the tree level by level. Building a single level takes constant time. In other words we have the following theorem.

Theorem 7.2 *All $C(n,r)$ combinations can be generated in $O(r)$ time with $C(n,r)$ processors on the CREW PRAM.*

Proof: We will prove the theorem by induction. We first prove that the time complexity is $O(r)$ by induction on r.

When $r = 1$, the algorithm takes only one step. Let us assume the algorithm for $r \leq k$ takes at most cr time for some constant c.

Now consider the case where $r = k+1$. In the algorithm, after taking a constant number of steps, there are $n-r+1$ recursive calls. All these calls execute simultaneously. Hence the time complexity is $T(k) + O(1) \leq c_1 + ck \leq c(k+1) = cr$, if $c_1 \leq c$. Hence our first claim is validated.

We next show again by induction on r that processor complexity is $O(C(n,r))$. For $r = 1$, clearly the number of processors needed are $n-r+1 = n-1+1 = n$. Hence the basis is proved since $C(n,1) = n$.

Let us assume the claim is valid for $r \leq k$.

For the case where $r = k+1$, there are $n-r+1$ recursive calls to *Gen_Comb* and by induction hypothesis the ith call needs $C(r-2+i, r-1)$ processors. Hence the number of processors needed for these recursive calls is the summation shown below.

$\sum_{i=1}^{n-r+1} C(r-2+i, r-1)$

This summation is nothing but the summation shown in Eq. (7.8) when the zero terms in that equation are neglected (see Eq. (7.3)).

Hence the value of the above summation is $C(n,r)$. Further, we need $n-r+1$ processors for the processing that is conducted before the recursive calls are invoked. So the number of processors needed are $max(n-r+1, C(n,r)) = C(n,r)$.

Hence our claim is validated. Further as in the algorithm, there are no simultaneous writes but simultaneous reads when a child reads the label of its parent, the algorithm can be implemented on the CREW PRAM. ∎

The number of r-permutations of n objects is $C(n,r)$ and each combination is of length r. So combination generation has a trivial lower bound of $O(rC(n,r))$ as we have to generate so much output. If we observe the *cost*[25] of our algorithm, it is $O(rC(n,r))$. Hence the above algorithm is cost optimal.

7.3 The r-Recursive Algorithm

7.3.1 Basic Algorithm

The main idea behind the algorithm is as follows: suppose we are interested in generating all the r-combinations of n objects in lexicographic order. We would like to realize r-combinations when $\frac{r}{2}$-combinations of n objects are available to us. So the algorithm is recursive. The termination of recursion occurs when $r = 1$ in which case obtaining the 1-combinations of n objects is trivial.

The algorithm in this subsection has the following restrictions:

Assumptions:

- r is a power of 2 and
- $r \leq \frac{n}{2}$

We relax these restrictions later (see Section 7.3.2).

The algorithm is given below formally.

```
Pre_Processing
//calculate C(p,q) where q ≤ r and for
1 ≤ p ≤ n.  Store these values in a two dimensional array, com.
for i = 1 to n*r in parallel do
```

```
    // group and shift are local variables of each processor
    group = (i-1) div r + 1;
    shift = (i-1) mod r + 1;
    val[(group-1) * r + shift] = group + shift - 1;
Calculate prefix multiplications of n segments of array val independently,
     where each segment is of length r and the i-th segment starts
     at (i-1)*r+1.
for q = 1 to r in parallel do
     for p = 1 to n in parallel do
         if q > p then com[p,q] = 0;
         else com[p,q] = val[(p-q+1-1)*r+q]/val[q]
```

```
Algorithm Generate_Combinations(n,r)
Begin
     if r == 1 then
         for i = 1 to n in parallel do
            A[i,1] = i;
     else
         Generate_Combinations(n, r/2);
         Double_Size();
End
Double_Size()
Begin
     Calculate for each r/2-combination its associated r/2-combination into array B
     Calculate Rank[i] for i = 1 to C(n,r/2) which is the rank
     of the corresponding first r-combination for each associated r/2-combination .
     Processor Allocation;
     Copy needed values;
End
Algorithm Processor_Allocation
Begin
```

```
    for i = 1 to C(n,r/2) in parallel do
        size[i] = com[n-A[i,r/2],r/2]*r;
        if size[i] = 0 then
           start[i] = (Rank[i] -1 ) * r + com[n-B[i,r/2], r/2]]*r + 1;
           end[i] = start[i] - 1;
        else
           start[i] = (Rank[i] -1 ) * r + 1;
           end[i] = start[i] + size[i] -1;
    p = C(n,r)*r;
    Generic_Processor_Allocation;
end
```

We first describe the pre-processing steps. In the pre-processing, we find the values of $C(p,q)$ when $q \leq r$ and for $1 \leq p \leq n$. We find the prefix multiplication of consecutive values of length r for each starting value from 1 to n. This has been accomplished by an application of standard prefix multiplication algorithm [19] on these different sets independently and simultaneously. Once these values are found, we find the values of $C(p,q)$ in constant time for the given ranges of p and q using Eq. (7.4).

The main algorithm is recursive in nature. The base case of the algorithm when $r = 1$ is trivial; the 1-combinations of n objects are simply the n objects in lexicographic order. The main procedure is *Double_Size()* which given $\frac{r}{2}$-combinations obtains the r-combinations.

The idea behind *Double_Size* is given an $\frac{r}{2}$-combination, we would like to obtain all the (possibly none) r-combinations with that $\frac{r}{2}$-combination as prefix.

We will illustrate the ideas through an example. Consider the 2-combinations of 8 objects $\{1,2\ldots,8\}$ shown below in lexicographic order (top to bottom, left to right).

(1 2) (2 3) (3 5) (4 8) .
(1 3) (2 4) (3 6) (5 6) .
(1 4) (2 5) (3 7) (5 7)
(1 5) (2 6) (3 8) (5 8)
(1 6) (2 7) (4 5) (6 7)
(1 7) (2 8) (4 6) (6 8)
(1 8) (3 4) (4 7) (7 8)

Consider the 4-combinations of 8 objects $\{1, 2 \ldots, 8\}$ shown below.

(1 2 3 4) (1 2 7 8) (1 4 6 7) (2 3 6 7) (3 4 5 7)
(1 2 3 5) (1 3 4 5) (1 4 6 8) (2 3 6 8) (3 4 5 8)
(1 2 3 6) (1 3 4 6) (1 4 7 8) (2 3 7 8) (3 4 6 7)
(1 2 3 7) (1 3 4 7) (1 5 6 7) (2 4 5 6) (3 4 6 8)
(1 2 3 8) (1 3 4 8) (1 5 6 8) (2 4 5 7) (3 4 7 8)
(1 2 4 5) (1 3 5 6) (1 5 7 8) (2 4 5 8) (3 5 6 7)
(1 2 4 6) (1 3 5 7) (1 6 7 8) (2 4 6 7) (3 5 6 8)
(1 2 4 7) (1 3 5 8) (2 3 4 5) (2 4 6 8) (3 5 7 8)
(1 2 4 8) (1 3 6 7) (2 3 4 6) (2 4 7 8) (3 6 7 8)
(1 2 5 6) (1 3 6 8) (2 3 4 7) (2 5 6 7) (4 5 6 7)
(1 2 5 7) (1 3 7 8) (2 3 4 8) (2 5 6 8) (4 5 6 8)
(1 2 5 8) (1 4 5 6) (2 3 5 6) (2 5 7 8) (4 5 7 8)
(1 2 6 7) (1 4 5 7) (2 3 5 7) (2 6 7 8) (4 6 7 8)
(1 2 6 8) (1 4 5 8) (2 3 5 8) (3 4 5 6) (5 6 7 8)

Let us consider r-combinations in lexicographic order of n objects. Note that lexicographic order is a *total order* [42] (i.e, for any two distinct r-combinations a and b either a lexicographically precedes b or *vice versa*). Suppose we consider the prefixes of length $\frac{r}{2}$ of r-combinations in order, then we can observe the following:

Lemma 7.3 *The prefixes of length i of r-combinations of $\{1, 2, \ldots, n\}$ in lexicographic order will be in lexicographic order when $i \leq r$ and adjacent duplicates are removed. More over, after adjacent duplicates removal these prefixes are the i-combinations of $n - r + i$ objects $S = \{1, 2, \ldots, n - r + i\}$.*

Proof: If these prefixes are not in lexicographic order, it is trivial to prove by contradiction that the original r-combinations are not in lexicographic order.

So, after adjacent duplicates removal each prefix of length i occurs only once. Let us put together these prefixes in order as they are occurring in the r-combinations.

We will prove that each of this prefix is a i-combination of S and that each i-combination of S must occur as exactly one prefix. Since it is already known that the prefixes are in lexicographic order, combining both the above statements results in proof of this part of the lemma.

Consider an arbitrary prefix $a_1 a_2 \ldots a_i$ of r-combinations. Then $a_i \leq n - r + i$, otherwise, $a_r > n$ in which case it is obvious that this is not a valid prefix. Recall that in a combination the values are in *strictly increasing* order. Since, $a_i \leq n - r + i$, it follows that the given prefix is an i-combination of S. Hence we have proved that every prefix is an i-combination of S.

Now, let us consider an arbitrary i-combination $(b_1 b_2 \ldots b_i)$ of S. Obviously, $b_i \leq n - r + i$. Attach the suffix $(c_1 c_2 \ldots c_{r-i})$ where $c_j = b_i + j$. It is easy to observe that the newly formed sequence is a valid r-combination. Hence, it follows that every i-combination of S must occur as a prefix for some r-combination of n objects $\{1, 2, \ldots, n\}$.

Hence, lemma follows. ∎

From this lemma, we can observe that for each $\frac{r}{2}$-combination there will be a set of r-combinations with this $\frac{r}{2}$-combination as prefix. It follows from the lemma that all these r-combinations will be together.

In the above example, with 1 2 as the prefix there are 15 combinations and more over all these prefixes are together and the suffixes are in lexicographic order. But it is interesting to observe that for some $\frac{r}{2}$-combinations there are no r-combinations with that $\frac{r}{2}$-combination as the prefix. For example while 6 8 is a 2-combination, there are no 4-combinations with 6 8 as the prefix.

Consider an $\frac{r}{2}$-combination $(a_1 a_2 \ldots a_{\frac{r}{2}})$. If $a_{\frac{r}{2}} + \frac{r}{2} > n$ then there will not be any r-combination with this $\frac{r}{2}$-combination as prefix. This can be easily seen as in a combination the values are increasing from left to right.

We can strengthen our lemma. Once an $\frac{r}{2}$-combination is given we want to characterize what are the suffixes that should be added in order that we can obtain the

corresponding r-combinations. This will be very much helpful in the design of algorithm.

Theorem 7.4 *Given p-combinations of n objects $\{1, 2, \ldots, n\}$ in lexicographic order and given any arbitrary prefix $c_1, c_2, \ldots, c_q$ which is a prefix of some p-combination there will be $C(n-c_q, p-q)$ p-combinations with this prefix. The suffixes that should be added to this prefix are $p-q$-combinations of $n-c_q$ objects $\{c_q+1, c_q+2, \ldots, n\}$. More over all these p-combinations formed by adding the aforesaid suffixes are in lexicographic order and are consecutive p-combinations.*

Proof: Consider the prefix $C = (c_1, c_2, \ldots, c_q)$. Obviously, this is a q-combination of n objects. Let us consider all p-combinations with this prefix. We now prove that all p-combinations with a given prefix will be together as consecutive. We prove by contradiction.

Suppose they are not consecutive. Then there exist three p-combinations B_1, B_2 and B_3 such that B_1, B_3 have the prefix $t_1 = (c_1 c_2 \ldots c_q)$ and B_2 has a different prefix t_2 of length q and B_1, B_2 and B_3 are in lexicographic order (not necessarily adjacent). Since t_1 and t_2 are different and B_1 precedes B_2 lexicographically it follows that t_1 precedes t_2 lexicographically. But this implies B_3 precedes B_2 lexicographically. A contradiction. Hence all the p-combinations with the same prefix are consecutive (or adjacent).

Now it is easy to seek that the suffixes that should be added to this prefix C are coming from the set $S = \{c_q+1, c_q+2, \ldots, n\}$. This follows from the definition of combinations. Further, whatever the suffixes we are appending to the prefix C are $(p-q)$-combinations of S, as in a combination, by definition, the values will be in increasing order. Hence it suffices to prove that all $C(n-c_q, p-q)$ $p-q$-combinations of S are used and appear in lexicographic order. If $p-q > n-c_q$ then there will not be any p-combinations with the given prefix, which contradicts assumption that it is a prefix of some p-combination. We can formally prove by contradiction. Suppose an arbitrary $p-q$-combination, D of S is missing. Now let us form a p-combination, by appending this suffix D of length $p-q$ to our given prefix of length q. This is a valid p-combination. But all valid p-combinations must appear in the lexicographic order. Hence it contradicts that a particular $p-q$-combination is missing.

So, from this we conclude that all the $p-q$ combinations of S are used for the given prefix. We have earlier proved that all the p-combinations with a given prefix will be

consecutive. Now to complete the proof we have to prove that the $p-q$ -combinations of S in lexicographic order will be attached to the given prefix, suffixes of the given prefix will be in lexicographic order.

This can also be proved by contradiction. Suppose that they are not in lexicographic order. There exists two p-combinations with the same above given prefix but whose suffixes are not in lexicographic order. That is one suffix does not lexicographically precedes the next one. But this contradicts that the p-combinations are in lexicographic order. Hence the theorem follows. ∎

As a special case of the above theorem we obtain

Corollary 7.5 *Let $(a_1 a_2 \ldots, a_{\frac{r}{2}})$ is an $\frac{r}{2}$-combination. The suffixes that should be added to this $\frac{r}{2}$-combination as prefix are nothing but the $\frac{r}{2}$-combinations of $\{a_{\frac{r}{2}}+1,\ldots,n\}$ in lexicographic order.* ∎

The above corollary needs a small qualification: if the cardinality of the set discussed above is less than $\frac{r}{2}$ then there are no r-combinations with the given prefix.

So, once we know the suffixes that should be added to each $\frac{r}{2}$-combination, it is easy to achieve our original goal - of forming the r-combinations.

The needed combinations useful as suffixes can be directly obtained from our original list of $\frac{r}{2}$-combinations itself. Our next lemma elaborates on this.

Lemma 7.6 *Given $C(n,r)$ r-combinations of n objects $\{1,2\ldots n\}$ in lexicographic order, the last $C(i,r)$ r-combinations are just the r-combinations of i objects $\{n-i+1,\ldots,n\}$ in lexicographic order for all i, where, $1 \leq i \leq n$.*

Proof: We will prove the lemma by induction on i. However, we will proceed from i to $i-1$.

As base case consider when $i = n$. Then it is obvious that our lemma is true. Assuming that the lemma is valid for the case $i = k$, we proceed to prove that it is valid for the case $i = k-1$.

Based on induction hypothesis, the last $C(k,r)$ r-combinations are just the r-combinations of k objects $\{n-k+1,\ldots,n\}$. Now consider these $C(k,r)$ r-combinations which are in lexicographic order. All of r-combinations with $n-k+1$ as the first value will be the

first combinations using Lemma 7.3. From Theorem 7.4, there will be $C(k-1,r-1)$ combinations with this prefix. If we consider the remaining combinations excluding these combinations there will be $C(k,r) - C(k-1,r-1) = C(k-1,r)$ combinations. But all these combinations are from the objects $\{n-k+2,\ldots,n\}$. Since we know that for $k-1$ objects there will be $C(k-1,r)$ r-combinations, it follows that we have all the r-combinations of these $k-1$ objects. Obviously all these will be in lexicographic order (otherwise, we will contradict that the original list of r-combinations is not in lexicographic order). ∎

From the above lemma, the suffixes that should be added to an $\frac{r}{2}$-combination will be coming from the given $\frac{r}{2}$-combinations itself.

To get the algorithm, we also have to look at the problem of processor allocation. Processor allocation is non-trivial as different $\frac{r}{2}$-combinations can have different number of r-combinations with them as prefixes. As we have seen we know *how many* processors we need for each $\frac{r}{2}$-combination but for efficient implementation we should know *which* processors are to be assigned to each $\frac{r}{2}$-combination. We can not use prefix sums as this will take unnecessarily long time as we are working with $C(n,\frac{r}{2})$ number of $\frac{r}{2}$-combinations.

The *ranking* of combinations achieves the same effect as that of prefix sums but with a much better time complexity. For each $\frac{r}{2}$-combination we would like to know which is the *first* r-combination with this $\frac{r}{2}$-combination as the prefix. The following lemma provides the answer.

Lemma 7.7 *Given an $\frac{r}{2}$-combination $(a_1 a_2 \ldots a_{\frac{r}{2}})$ which is a valid prefix for the r-combinations, the first r-combination with this prefix is $(a_1 a_2 \ldots a_{\frac{r}{2}} a_{\frac{r}{2}}+1 a_{\frac{r}{2}}+2 \ldots a_{\frac{r}{2}}+\frac{r}{2})$.*

Proof: From Corollary 7.5, we know that the suffixes that should be attached to the above $\frac{r}{2}$-combination are just the $\frac{r}{2}$-combinations of the $n - a_{\frac{r}{2}}$ objects $\{a_{\frac{r}{2}}+1, a_{\frac{r}{2}}+2,\ldots,n\}$. But observe that the first $\frac{r}{2}$-combination for the above list of objects is $(a_{\frac{r}{2}}+1 a_{\frac{r}{2}}+2 \ldots a_{\frac{r}{2}}+\frac{r}{2})$. Hence our lemma follows. ∎

Now once we know the first r-combination corresponding to each *applicable* $\frac{r}{2}$-combination (some $\frac{r}{2}$-combinations are not applicable as prefixes to the r-combinations), we find its rank.

We can find the rank of an r-combination in $O(\log r)$ time using $r/\log r$ processors, as we have to sum r values (see Section 2.2.2).

Processor Allocation

For each $\frac{r}{2}$-combination which is applicable as a valid prefix for r-combination we find rank of the first r-combination with this $\frac{r}{2}$-combination as prefix. We also find for each $\frac{r}{2}$-combination the number of r-combinations with this prefix.

We first would like to characterize a valid prefix in a more general setting.

Lemma 7.8 *A r-combination of n objects $S = \{1, 2, \ldots, n\}$, $A = (a_1 a_2 \ldots a_r)$ is a valid prefix for some k-combination ($k \geq r$) of S if and only if $a_i \leq n - k + i$ for all $1 \leq i \leq r$.*

Proof: The if part is simple since if $a_r \leq n - k + r$ then A is also a r-combination for $n - k + r$ objects $\{1, 2, \ldots, n - k + r\}$. But by Lemma 7.3, every r-combination of $n - k + r$ objects $\{1, 2, \ldots, n - k + r\}$ is a valid prefix for some k-combination.

The only if part is also equally simple. If A is a valid prefix for k-combination then it must be the case that $a_i \leq n - k + i$. Otherwise, we can not form a valid k-combination as the elements are strictly increasing in a combination. Thus, if $a_i > n - k + i$ for some i, then a_k must be greater than n. Then, it follows that A is not a valid prefix. ∎

From the lemma it follows that some $\frac{r}{2}$-combinations are not valid prefixes for r-combinations. But even in such a case, we must know, how many processors are used by the valid $\frac{r}{2}$-combinations that come before it. This is necessary for the our processor allocation procedure (see Section 2.3). To facilitate this case and to make the approach more uniform we define *associated* combination. Each $\frac{r}{2}$-combination has got an associated $\frac{r}{2}$-combination, which is the same as the $\frac{r}{2}$-combination if it is a valid prefix for r-combinations. If it is not a valid prefix, the associated $\frac{r}{2}$-combination for the given $\frac{r}{2}$-combinations is the *nearest lexicographically preceding* $\frac{r}{2}$-combination which is a valid prefix for r-combinations. The calculation of the associated $\frac{r}{2}$-combination for an $\frac{r}{2}$-combination is facilitated by the following lemma which is given in more general form because of its use in a later section.

Lemma 7.9 *Given an r-combination, $A = (a_1 a_2 \ldots a_r)$ of n objects $S = \{1, 2, \ldots, n\}$, the nearest lexicographically preceding (or same) r-combination which can act as the*

prefix of k-combination ($k \geq r$) of S, is given by $(b_1 b_2 \ldots b_r)$ where $b_i = n - k + i$ if $a_i > n - k + i$ else $b_i = a_i$.

Proof: In $B = (b_1 b_2 \ldots b_r)$, if $b_j < a_j$, then $b_l = n - k - l$, for all $j \leq l \leq r$, as if $a_i > n - k + i$ then $a_{i+1} > n - k + i + 1$ (since in a combination the values are in strictly increasing order).

So, let us denote by m the maximum integer such that $a_j = b_j$ for all $1 \leq j \leq m$. Note that $b_j = n - k + j$ for $m + 1 \leq j \leq r$.

Using Lemma 7.8, if $C = (c_1 c_2 \ldots c_r)$ is a valid prefix for k-combination then it must be the case that $c_i \leq n - k + i$ for $1 \leq i \leq r$.

We observe that, B is a valid prefix for k-combination and more over $b_i \leq a_i$ for all $1 \leq i \leq n$, and hence B lexicographically precedes A or is same as A.

Obviously if $A = B$, then our lemma is true.

So, let us consider the case below where $A \neq B$. So, in that case, B lexicographically precedes A.

we prove the lemma by contradiction. Let us assume that B is not the nearest valid prefix for A and $D = (d_1 d_2 \ldots d_r)$ is the correct one. Then, $B \prec D \preceq A$.

Since, $a_j = b_j$ for $1 \leq j \leq m$, it follows that $d_j = a_j$ for $1 \leq j \leq m$. Let us assume m_1 the minimum index for which $b_j < d_j$. Such an index must exist since B lexicographically precedes D. Obviously $m_1 > m$. Note that as $b_{m_1} = n - k + m_1$, $d_{m_1} > n - k + m_1$ which implies that D is not a valid prefix for k-combination by Lemma 7.8. Hence a contradiction. So, the lemma is proved. ∎

By the above lemma it is apparent that, we can obtain the associated $\frac{r}{2}$-combination for each $\frac{r}{2}$-combination in constant time with $\frac{r}{2}$ processors.

We need to allocate $rC(n, r)$ processors for building the r-combinations as there are $C(n, r)$ r-combinations. For each of these processors we need to allocate work. For each $\frac{r}{2}$-combination we need to find all the r-combinations with this $\frac{r}{2}$-combination as prefix. Hence there is a task corresponding to each $\frac{r}{2}$-combination. For each processor we need to inform to which $\frac{r}{2}$-combination does it belong to. We also need to know the index of each processor with respect to the processors that are allocated to each $\frac{r}{2}$-combination. Using this information, it is easy to copy the respective information to build r-combinations.

In our algorithm we indicated how we obtain this information by using an application of generic processor allocation.

Using the details available from processor allocation a processor p proceeds as follows. Let $s = (p-1)\ div\ r + 1$. Then processor p will be working towards obtaining the s-th r-combination in lexicographic order. Suppose p belongs to $G[p] = a$-th $\frac{r}{2}$-combination. Then processor p is the $N[p] = b$ th processor that are allocated to a-th $\frac{r}{2}$-combination in order. Assume a th $\frac{r}{2}$-combination is the prefix for c r-combinations. Let $d = (b-1)\ div\ r + 1$. Then p-th processor should work to form d-th r-combination with a-th $\frac{r}{2}$-combination as prefix. Let $e = (b-1)\ mod\ r + 1$. Then p-th processor should obtain the e-th item of the r-combination on which it is working. If $e \leq \frac{r}{2}$ then p-th processor will obtain the e-th item of $\frac{r}{2}$-combination to which it belongs and writes it in the correct position. If $e > \frac{r}{2}$ then it needs to obtain the value from the end of the $\frac{r}{2}$-combinations depending on its position.

To accomplish this we need to know to which $\frac{r}{2}$-combination each processor belongs. First we obtain the ranks of the $\frac{r}{2}$-combinations which act as the first r-combination with that prefix. Consider w-th arbitrary $\frac{r}{2}$-combination. Let its rank as the prefix of the first r-combination be u. Similarly let us assume that the number of r-combinations with this $\frac{r}{2}$-combination as prefix be v. Then the processors numbered $(u-1)r+1$ to $(u-1)r+vr$ should be allocated for this $\frac{r}{2}$-combination. On the other hand, for those $\frac{r}{2}$-combinations which are not valid prefixes for r-combinations, we have obtained its associated $\frac{r}{2}$-combination and using the associated combination, we obtain the values of arrays *start*, *end* and *size* for generic processor allocation. Using this information, we apply generic processor allocation (see Section 2.3). So each processor knows to which $\frac{r}{2}$-combination it belongs to and what is its index among the processors allocated to that $\frac{r}{2}$-combination.

We now analyze the algorithm.

First look at the preprocessing. In the first step we are initializing tables which needs constant time and nr processors. In the next step we are finding the prefix multiplications of a number of segments of equal length simultaneously using the standard algorithm for prefix multiplication[25]. It has a time complexity of $O(\log r)$ as each segment is of length r. The total work done is $O(nr)$ as the prefix multiplication algorithm is optimal.

Once this processing is over, we find the values of $C(p,q)$ where p varies from 1 to n and q varies from 1 to r. This is calculated using Eq. (7.4) and can be done in constant time with nr processors as it involves retrieval of two values and their division. Thus, the total time for the pre-processing is $O(\log r)$ with a work of $O(nr)$.

Now we analyze the time complexity and work of main part of our algorithm. Recall that r is a power of 2 and $r \leq \frac{n}{2}$. Let $T(n,r)$, $W(n,r)$ denote the time complexity and work done when we are generating r-combinations of n objects respectively.

The algorithm is recursive and the main part involves the analysis of *Double_Size()*. *Double_Size* obtains the r-combinations from $\frac{r}{2}$-combinations. We assume that the $\frac{r}{2}$-combinations are available in an array. We initially find the number of r-combinations that are associated with each $\frac{r}{2}$-combination. This can be done in constant time with $C(n,\frac{r}{2})$ processors. Then we find the associated $\frac{r}{2}$-combination for each $\frac{r}{2}$-combination in constant time. After that for each $\frac{r}{2}$-combination, we find the rank of the *first* r-combination with its associated $\frac{r}{2}$-combination as prefix. This calculation involves $C(n,\frac{r}{2})$ independent summations and can be done in $O(\log r)$ time with a work of $O(C(n,\frac{r}{2}).r)$.

Once the calculation of ranks is over, we do the processor allocation. From Section 2.3, this can be done in $O(\log\log*(C(n,\frac{r}{2}) + C(n,r).r))$ time with $O(C(n,r).r + C(n,\frac{r}{2}))$ work on CREW PRAM. Once this is done each processor copies the needed value from the $\frac{r}{2}$-combinations to form the r-combinations in constant time with $O(C(n,r)r)$ processors.

When $r = 1$, the algorithm takes constant time and n processors. Thus, we form the following recurrence relations.

$$T(n,r) = \begin{cases} T(n,\frac{r}{2}) + O(\log\log*(C(n,\frac{r}{2}) + C(n,r).r)) + O(\log r) & \text{if } r > 1 \\ c_1 & \text{if } r = 1 \end{cases}$$

where c_1 is a constant.

The above equation can be simplified by observing that $C(n,r).r = O(n^{r+1})$. Hence $O(\log\log*(C(n,\frac{r}{2}) + C(n,r).r)) = O(\log\log*(n^{r+1})) = O(\log(\log*(r+1) + \log*n)) = O(\log\log*r + \log\log*n)$. Thus,

$T(n,r) = T(n,\frac{r}{2}) + c_2.\log r + c_3.\log\log*n.$

where c_2 and c_3 are constants.

Our algorithm employs induction on r and hence the algorithm goes $\log r$ levels deep. Using this[13]

$T(n,r) = O(\log r(\log r + \log\log * n))$

For work performed by the algorithm, we can obtain the following recurrence relation:

$$W(n,r) = \begin{cases} W(n,\frac{r}{2}) + k_1.C(n,r)r & \text{if } r > 1 \\ k_2.n & \text{if } r = 1 \end{cases}$$

where k_1, k_2 are constants.(Note that we used Eq. (7.5))

We formally show below that $W(n,r) \leq 2k_1.C(n,r).r$

Proof: The proof is based on induction on r.

As base case consider when $r = 1$. From the algorithm we know that $W(n,1) = k_2.n \leq 2k_1.n = 2k_1.C(n,1)$ if $2k_1 > k_2$.

For the induction hypothesis, let us assume $W(n,r) \leq 2k_1.C(n,r).r$ for $r \leq 2^k$. Let us consider the case $2r$

$W(n,2r) = W(n,r) + k_1.C(n,2r).2r \leq 2k_1.C(n,r).r + k_1.C(n,2r).2r = k_1.C(n,r).$ $k_1.C(n,2r).2r \leq k_1.C(n,2r).2r + k_1.C(n,2r).2r$

since according to our restrictions, $2r \leq \frac{n}{2}$ and using Eqs. (7.5) and (7.6).

Thus, $W(n,2r) \leq 2k_1.C(n,2r).2r$

Hence our claim follows. ∎

Generation of r-combinations has a lower bound of $O(C(n,r).r)$. So our algorithm is optimal.

Putting all these together leads to the following theorem.

Theorem 7.10 *Algorithm works with a time complexity of* $O(\log^2 r + \log r.\log\log * n)$ *with optimal work and on CREW model when* r *is a power of* 2 *and* $r \leq \frac{n}{2}$. ∎

On the other hand, if we use COMMON CRCW PRAM then Theorem 2.4 will be applicable for processor allocation and we can show that,

Theorem 7.11 *Algorithm works with a time complexity of* $O(\log r(\log r + \alpha(n))$ *with optimal work and on COMMON CRCW model when* r *is a power of* 2 *and* $r \leq \frac{n}{2}$. ∎

7.3.2 Generalization of the Algorithm

In the above section, we have seen that the r-combinations will be generated only when r is a power of 2. In this section, we generalize that algorithm to the case where r is not a power of 2. But the other restriction $r \leq \frac{n}{2}$ remains. This restriction, we will remove in Section 7.3.3.

This generalized algorithm uses the original algorithm. The approach that is used in the generalized algorithm in its basic form is same as that of the restricted algorithm. The algorithm is given below.

```
Pre-Processing
Calculate the values of C(p,q) for 0 ≤ p ≤ n, 0 ≤ q ≤ r.
Algorithm Generate_Combinations_Generalized(n,r)
Begin
   r1 is the largest integer such that
        a) r1 ≤ r and
        b) r1 is a power of 2.
        c) r1 + r2 = r.
   Generate_Combinations(n,r1);
   // Let these r1-combinations be stored in the array A[1..C(n,r1), 1..r1]
   q = r1 − r2;
   Store the suffixes of length r2 of the first C(n − q, r2)
   r1-combinations in the array B[1..C(n − q, r2), 1..r2];
   Increase_Size();
End
Increase_Size()
Begin
   // Increases the size of the r1-combinations to become r-combinations.
   Obtain the associated r1-combination for each r1-combination;
   Calculate the ranks of associated r1-combination as first r-combination
   Calculate the number of processors;
   Processor Allocation;
   Copy needed values;
```

End

Let r_1 be the largest power of 2 less than or equal to r. So, r_1 is the value of the most significant bit of r.

Based on the value of r_1, we can make the following observations.

Observations

1. $r_1 \leq r$
2. $r_1 \geq \lceil \frac{r}{2} \rceil$
3. $r_2 \leq \lfloor \frac{r}{2} \rfloor$

We generate r_1-combinations of n objects using our earlier algorithm. These r_1-combinations are stored in a two dimensional array A. Now we obtain r-combinations using these r_1-combinations. In our earlier algorithm, we used *Double_Size()* to achieve this. In this case we use *Increase_Size()* for similar purpose.

If we observe r_1-combinations, each of r_1-combination act as a prefix for some number (possibly none) of r-combinations. All prefixes of length r_1 of the r-combinations are in lexicographic order by Lemma 7.3. More over each prefix is a valid r_1-combination. Hence we essentially add the suffixes of length r_2 to these r_1-combinations to make them r-combinations. Further, these suffixes that need to be added to these r_1-combinations can be obtained from these r_1-combinations itself. Consider an arbitrary r_1-combination, $(d_1 d_2 \ldots d_{r_1})$. The suffixes that should be added to this r_1-combination are r_2 combinations of $n - d_{r_1}$ objects *viz.* $\{d_{r_1}+1, d_{r_1}+2, \ldots, n\}$ by Theorem 7.4. If $n - d_{r_1} < r_2$ then there are no r-combinations associated with this prefix. So for each r_1-combination, we need to obtain r_2-combinations for some set of objects. These set of objects are always of the form $\{s_1, s_1 + 1, \ldots, n\}$ for some arbitrary s_1. The minimum s_1 occurs when the r_1-th value of a r_1-combination is minimum. The minimum r_1-th value that is possible for an r_1-combination is r_1. This corresponds to the first r_1-combination *viz.* $(12 \ldots r_1)$. This r_1-combination needs r_2-combinations from the set of objects $\{r_1+1, r_1+2, \ldots, n\}$. The r_2-combinations for other set of objects $\{t_1, t_1 + 1, \ldots, n\}$ can be obtained from the above set of objects provided $t_1 \geq r_1 + 1$. This follows from Lemma 7.6. So, if we some how get the r_2-combinations of $n - r_1$ objects $\{r_1 + 1, r_1 + 2, \ldots, n\}$ then our

work is essentially done. Note that by Lemma 7.6, r_2-combinations of the set of objects $\{v_1, v_1+1, \ldots, n\}$ also suffices when $v_1 \leq r_1 + 1$.

Let $q = r_1 - r_2$. Let us consider the r_1-combinations. The first r_1-combination is $(12 \ldots r_1)$. There will be $C(n-q, r_2)$ r_1-combinations with prefix $1, 2, \ldots, q$. More over, by Theorem 7.4 they are just the r_2-combinations of $n-q$ objects $\{q+1, q+2, \ldots, n\}$. Further by Theorem 7.4, the first $C(n-q, r_2)$ combinations have the same prefix $1, 2 \ldots, q$ of length q. Hence these $C(n-q, r_2)$ suffixes of length r_2 correspond to the r_2-combinations of $n-q$ objects $\{q+1, q+2, \ldots, n\}$. As $q < r_1$, these are the needed set of r_2-combinations. So all the r_2-combinations needed can be obtained. These suffixes are stored in an array B. Using Lemma 7.6, we can obtain the needed set of r_2-combinations for any set of objects. More specifically, suppose we are interested in obtaining the r_2-combinations of objects $\{w, w+1, \ldots, n\}$. Assume that $w \geq q+1$. Then the needed set of r_2-combinations can be obtained by taking the last $C(n-w+1, r_2)$ combinations from the above set of r_2-combinations.

Now we have the essential information needed to form the r-combinations. After this, we proceed in manner similar to that of our earlier algorithm. This details of *Increase_Size*() are very similar to *Double_Size*().

Let $T(n, r)$, $W(n, r)$ denote the time complexity and the work done by the generalized algorithm respectively.

We invoke the earlier algorithm for generating combinations with parameters n, r_1. Hence based on Theorem 7.10, this takes a time of $O(\log r_1(\log r_1 + \log\log * n))$ and a work of $O(C(n, r_1).r_1)$.

We then invoke *Increase_Size* to obtain the r-combinations from these r_1-combinations. This takes a time of $O(\log r + \log\log * n)$ and work of $O(C(n, r_1).r + C(n, r).r)$. Thus,

$T(n, r) = c_1.\log r_1(\log r_1 + \log\log * n) + c_2.(\log r + \log\log * n)$ where c_1 and c_2 are constants. Since $r_1 \leq r$, $r_1 \geq \frac{r}{2}$ and $r \leq \frac{n}{2}$,

$T(n, r) = O(\log r(\log r + \log\log * n))$

The work done by the algorithm is:

$W(n, r) = c_3.C(n, r_1).r_1 + c_4.C(n, r_1).r + c_5.C(n, r).r$ where c_3, c_4 and c_5 are constants.

Here, $W(n, r) = O(C(n, r).r)$

So we obtain the following theorem

Theorem 7.12 *Algorithm optimally generates the r-combinations of n objects in lexicographic order with a time complexity of $O(\log r(\log r + \log\log * n))$ on CREW PRAM when $r \leq \frac{n}{2}$.* ∎

Using Theorem 2.4 for processor allocation, we can show that

Theorem 7.13 *Algorithm optimally generates the r-combinations of n objects in lexicographic order with a time complexity of $O(\log r(\log r + \alpha(n)))$ on COMMON CRCW PRAM when $r \leq \frac{n}{2}$.* ∎

7.3.3 Generalization of the Algorithm (Case $r > \lceil \frac{n}{2} \rceil$)

Till now we have assumed that $r \leq \frac{n}{2}$ in our algorithm. We next remove this restriction. Suppose that we are interested in generating r-combinations of n objects where $r > \frac{n}{2}$.

We give an algorithmic interpretation to Eq. (7.2). If we are interested in generating r-combinations, then we can initially generate $n-r$-combinations of n objects; each $n-r$-combination represents the objects we have selected. But if we consider the objects we have not selected they will be r objects corresponding to each $n-r$-combination. In other words there is a one-to-one correspondence between each r-combination and $n-r$-combination. Instead of generating r-combinations we first generate $n-r$-combinations using above algorithm. Since $r > \frac{n}{2}$, $n - r < \frac{n}{2}$ and above algorithm is applicable. Once we have generated the $n-r$-combinations, we will generate r-combination corresponding to each $n-r$-combination.

For each $n-r$-combination, we allocate an array of length n. In that array we put 1's for each object that are selected and 0's for them that are not selected. So, we bring together the objects that are not selected by a single application of prefix sum. However, the resulting r-combinations may not be in lexicographic order. But fortunately they are in decreasing lexicographic ordering. This is proved in Lemma 7.14.

Lemma 7.14 *Given $n-r$-combinations of n objects in lexicographic order, the corresponding r-combinations will be in decreasing lexicographic order.*

Proof: Let P_1 and P_2 be arbitrary *consecutive* $n-r$-combinations in lexicographic order. Let Q_1 and Q_2 be their corresponding r-combinations. Then we will prove that $Q_2 \prec Q_1$; the lemma follows as lexicographic ordering is a total ordering.

First we look at the problem of determining the next combination given an arbitrary combination.

Let $P_1 = (a_1 a_2 \ldots a_{n-r})$ and $P_2 = (b_1 b_2 \ldots b_{n-r})$.

Then suppose i is the *maximum* index for which $a_i < n-(n-r)+i$ for $1 \leq i \leq n-r$. Then $b_j = a_j$ for $1 \leq j \leq i-1$, $b_i = a_i + 1$ and $b_j = b_{j-1} + 1$ for $i+1 \leq j \leq n-r$[33] (i must exist since P_1 is *not* the last $(n-r)$-combination). Moreover, since $a_i < n-(n-r)+i$ then $a_{i+1} = n-(n-r)+i+1$ and hence $a_{i+1} - a_i \geq 2$.

Let us consider the formation of Q_1 and Q_2 from P_1 and P_2 respectively. Recall that in a combination the objects are in increasing order. Let $Q_1 = (c_1 c_2 \ldots c_r)$ and $Q_2 = (d_1 d_2 \ldots d_r)$.

Since $b_j = a_j$ for j varying from 1 to $i-1$, it follows that $c_k = d_k$ whenever $c_k < a_i$. Let m be the largest such k. Since $b_i = a_i + 1$. Now $d_{m+1} = a_i$ but $c_{m+1} > d_{m+1}$. Hence, Q_2 lexicographically precedes Q_1. Hence our lemma follows. ∎

Based on this lemma it is easy to lexicographically order r-combinations; just put i th combination at $C(n,r) - i + 1$ place.

We analyze the algorithm. We use the usual notation for the time and work.

For generation of r-combinations of n objects in this case, we initially obtain the $n-r$-combinations of n objects. This takes a time of $O(\log(n-r)(\log(n-r) + \log\log * n)$ and a work of $O(C(n,n-r).(n-r))$ by Theorem 7.12. After this we take a time of $O(\log n)$ and a work of $O(C(n,n-r).n)$. Thus,

$T(n,r) = c_1.\log(n-r)(\log(n-r) + \log\log * n) + c_2.\log n$

or, $T(n,r) = O(\log r(\log r + \log\log * n))$

since $r > \frac{n}{2}$, $n = O(r)$.

Similarly,

$W(n,r) = c_3.C(n,n-r).(n-r) + c_4.C(n,n-r).n$

where c_3 and c_4 are constants.

Since $C(n,r) = C(n,n-r)$ and $r > \frac{n}{2}$,

$W(n,r) = O(C(n,r).r)$

So combining all of our cases, we obtain the following theorem.

Theorem 7.15 *We can generate all the r-combinations in lexicographic order and optimally with a time complexity of $O(\log r(\log r + \log\log * n))$ on the CREW PRAM.* ∎

Using theorem 2.4 for processor allocation, we can show that

Theorem 7.16 *We can generate all the r-combinations lexicographically and optimally with a time complexity of $O(\log r(\log r + \alpha(n)))$ on the COMMON CRCW PRAM.* ∎

7.4 The n-Recursive Algorithm

In this section, we discuss an algorithm to generate combinations with recursion on n.

7.4.1 Basic Algorithm

Like our first algorithm this algorithm is also based on giving algorithmic interpretation to a well known combinatorial identity. This reinforces our belief that algorithmic interpretation of combinatorial identities provides a good basis through which we can design parallel algorithms.

Our algorithm is based on *Vandermonde's Convolution* [26],

$$\sum_k C(m_1,k)C(m_2,r-k) = C(m_1+m_2,r) \tag{7.9}$$

We give a combinatorial interpretation[40] to this identity. Suppose we are interested in forming a team of r members out of $m_1 + m_2$ persons. We would like to know how many such teams we can form. Out of these persons m_1 are male and m_2 are female. The teams can be formed in the following ways. We can select k males out of the m_1 males and $r-k$ females out of the m_2 females. We can select k males out of m_1 males in $C(m_1,k)$ ways and $r-k$ females out of m_2 females in $C(m_2,r-k)$ ways. Since both these possibilities are independent, there are $C(m_1,k)C(m_2,r-k)$ possible teams for a given value of k. We get different possibilities based on different values of k. All teams that are formed are different. This exhausts all the possibilities. The left part of the

identity specifies the number of possible teams that we can form. The identity follows as we can form a team of r members out of $m_1 + m_2$ persons in $C(m_1 + m_2, r)$ ways.

We use a special case of this identity when $m_1 = \frac{n}{2}$ and $m_2 = \frac{n}{2}$ shown below (assume n is even).

$$\sum_{k=0}^{r} C(\frac{n}{2}, k)C(\frac{n}{2}, r-k) = C(n, r) \tag{7.10}$$

We can give the following algorithmic interpretation to the above identity. Suppose we are interested in generating r-combinations of n objects. Then the above identity specifies that we can select k objects out of first $\frac{n}{2}$ objects in $C(\frac{n}{2}, k)$ ways and select the remaining $r-k$ objects from the other set of $\frac{n}{2}$ objects in $C(\frac{n}{2}, r-k)$ ways. Since both these are independent and we have to do both the operations, this can be done in $C(\frac{n}{2}, k).C(\frac{n}{2}.r-k)$ ways. For different values of k this leads to different selections. All these are disjoint. Hence these exhaust all the possible r-combinations that we can obtain.

In the above interpretation, while we are interested in generating r-combinations of n objects, we have to solve subproblems where we have to select k objects out of $\frac{n}{2}$ objects for varying values of k. So, here the recursion is on n. The number of recursive calls depend on the value of r at a particular level of recursion.

We make the following assumptions in this section.

Assumptions:

- n must be a power of 2.
- $r \leq \frac{n}{2}$

We will relax these restrictions in Section 7.4.2.

The identity in Eq. (7.10) can be made more precise by eliminating zero terms using Eq. (7.3).

$$\sum_{k=0}^{r} C(\frac{n}{2}, k).C(\frac{n}{2}, r-k) = C(n, r),\ if\ r \leq \frac{n}{2} \tag{7.11}$$

$$\sum_{k=r-\frac{n}{2}}^{\frac{n}{2}} C(\frac{n}{2}, k).C(\frac{n}{2}, r-k) = C(n, r),\ if\ r > \frac{n}{2} \tag{7.12}$$

Each term in the identity (7.10) is a product of two binomial coefficients and hence represents two recursive calls. If one of them is zero then we have unnecessarily generated both the recursive calls whose result we are not going to use. But as this consumes resources, we have removed all the zero terms. In Eqs. (7.11), (7.12), we have two recursive calls for each term and they are useful in the final computation *i.e*, the generation of r-combinations.

We next study what are the specific recursive calls that will be made in the recursive algorithm based on the above identities. (Note that when we generate p-combinations of some set of q objects, we generically call it as a recursive call for p-combinations of q objects.)

Lemma 7.17 *Assume we are generating the r-combinations of n objects, where $r \leq \frac{n}{2}$ and n is a power of 2, using the recursive algorithm based on Eqs. (7.11) and (7.12). As part of the recursive algorithm we generate p-combinations of m objects if and only if m is a power of 2, $m < n$ and $p \leq min(m, r)$.*

Proof: Since n is a power of 2, let $n = 2^s$ for some integer s. We prove the lemma by induction on s and go from s to $s - 1$.

Note that the recursive algorithm is based on divide and conquer and based on the identities (7.11) and (7.12). It is obvious that we will be calculating combinations of m objects where m is a power of 2 and $m < n$. This holds for all number of objects of size 2^i where i varies from 0 to $s - 1$.

So, our main task is proving that the recursive calls are made for the p-combinations where p is as indicated in Lemma.

Let us consider the if part of lemma. As basis, we consider the case when we are generating combinations for $\frac{n}{2}$ objects. In fact these are first set of recursive calls that are made. Since $r \leq \frac{n}{2}$, Eq. (7.11) is applicable and we will be generating p-combinations of $\frac{n}{2}$ objects where $0 \leq p \leq r$.

As part of induction hypothesis, we assume that the Lemma is valid for the case of $\frac{n}{2^q}$ objects. Here $q \leq s$. That is we generate p-combinations for $\frac{n}{2^q}$ objects when $0 \leq p \leq min(r, \frac{n}{2^q})$. Now we have to prove that the lemma is valid for case of $\frac{n}{2^{q+1}}$ objects. That is we generate the p-combinations for the number of objects where p varies from 0 to $min(r, \frac{n}{2^{q+1}})$.

For arbitrary t, consider the t-combinations of $\frac{n}{2^{q+1}}$ objects where t lies in the range 0 to $min(r, \frac{n}{2^{q+1}})$.

Since $\frac{n}{2^q} > \frac{n}{2^{q+1}}$ it follows that t definitely lies in the range 0 to $min(r, \frac{n}{2^q})$. From induction hypothesis we know that we generate t-combinations of $\frac{n}{2^q}$ objects. But let us consider the recursive call correspond to this generation of combinations. We observe that since $t \leq min(r, \frac{n}{2^{q+1}}), t \leq \frac{n}{2^{q+1}}, i.e, t \leq \frac{1}{2}\frac{n}{2^q}$, it follows that in the generation of t-combinations of $\frac{n}{2^q}$ objects we will be using the Eq. (7.11) where $t = r$. But if we observe the first term in that identity, we find that we need to calculate the t-combinations of $\frac{n}{2^{q+1}}$ objects. Since t we have chosen is arbitrary, the proof of the first part of lemma is complete.

Note that in identities (7.11) and (7.12), all terms are non-zero which implies that each recursive call produces non-zero number of combinations.

We next prove the only if part of Lemma. This can be easily proved, as in generation of r-combinations we never generate p-combinations of any number of objects when $p > r$. If $p > u$ then we never generate p-combinations of u objects. Hence our lemma follows. ∎

As the identity indicates, we have a number of recursive calls for the generation of r-combinations. We have to use parallel recursion, where the number of recursive calls will be more than one. We avoid parallel recursion by implementing the algorithm bottom-up. We first execute all the calls where we generate i-combinations for 1-object. Then we generate the combinations for 2-objects and so on until we generate the r-combinations of n-objects.

In the bottom-up approach we will be obtaining i-combinations for varying values of i for the same set of objects simultaneously. But the number of processors that we need for each call are not same. Hence the necessity to determine which processor do which work. So, in the pre-processing for the algorithm, we will determine what processor will do which work for the *entire* main processing. The algorithm is given in Fig. 3.

Pre-processing

In main procedure we have to calculate binomial coefficients at a number of places. We will calculate $C(p,q)$ for $1 \leq p \leq n$ and $0 \leq q \leq r$ and store them. This is done in pre-processing as follows. Apply prefix multiplications[25] on the array A already

initialized with $A[i] = i$ for $1 \leq i \leq n$. After prefix multiplication, $A[i] = i!$. So, once we have calculated the factorial values, we use Eq. (7.1) calculate the value of binomial coefficients.

Once the calculation of binomial coefficients is over, we do processor allocation.

We now introduce notation which will be used in our presentation. We refer the generation of combinations for a particular number of objects as one level. The ith level corresponds to the generation of combinations for 2^i objects. So, there are in total $\log n + 1$ levels from 0 to $\log n$. We refer the level where we are generating the combinations for n objects as the *top* level. We also refer the level where we are generating the combinations for 1 object as the *bottom* level.

All the calls related to a particular level of recursion will execute simultaneously. The processors from one level will be used in the next level.

The processor allocation for a particular call will be based on the Eqs. (7.11) and (7.12). The above equations provides us with details of how to form the given set of combinations for a given set of objects and also the number of processors needed and how we should distribute them. But we should recall that the Eqs. (7.11) (7.12) specifies only the number of combinations that we will be obtaining but *not* the number of processors that we should devote. We will be allocating p processors for each p-combination. If we multiply by p for the equation to generate p-combinations that will suffice in terms of the needed processors. That is for generating the p-combinations of k objects, we will be allocating the number of processors for each term such that we will have *one* processor for each *item* of a combination.

Of $\log n + 1$ levels, we consider the top level and the bottom level separately. The remaining levels will be considered in a uniform fashion. For all the levels except the top level, we generate all the p-combinations for p varying from 0 to $min(m, r)$ where we want to generate these p-combinations for m objects.

For each level of recursion, we will try to find the number of processors needed for p-combinations of m objects where p varying from 1 to r instead of p varying from 1 to $min(m, r)$. The modified range for p simplifies the processing with no change in work, for those p which are outside the allowed range, no processors will be allocated.

There are $\log n - 1$ levels for which we have to do processor allocation – level 1 to

level $\log n - 1$. In each of these levels, there will be r recursive calls. We create an array C of length $(\log n - 1)r(r+1)$ such that the locations $(i-1)r(r+1)+1$ to $ir(r+1)$ correspond to level i. Suppose level i starts at $START$ then the processors allocation details for the jth call in level i correspond to locations $START + (j-1)(r+1)$ to $START + j(r+1) - 1$. These locations will be filled based on Eq. (7.10). Note that even though we are not using Eqs. (7.11), (7.12), the *extra* locations will contain zeroes and hence no processors will be allocated to them.

We apply prefix sums on this array C. Let the result be in D. We consider each location of D as specifying a task to be done, with the needed number of processors as specified in the array after prefix summation. We then apply generic processor allocation (see Section 2.3). The calculation of the arrays needed like *start*, *end* and *size* and the value p are specified in the algorithm itself and are self explanatory.

Once generic processor allocation is done, we will get the arrays G and N from it. From now onwards, we refer the array G as *Alloc* and the variable p by $Total$ which denotes the total number of processors that are used in the $(\log n - 1)$ levels.

Let us assume that d_i denotes the number of processors we need for level i. More over let e_i's denote the prefix sums on this array d_i. We calculated e_i as part of obtaining array D. The first d_1 locations of the array *Alloc* provide the processor allocation details for level 1. Similarly the next d_2 locations provide the processor allocation details for level 2 and so on. In other words, the locations of *Alloc* array, $e_{i-1}+1$ to e_i provide the processor allocation details for the level i(Assume $e_0 = 0$.). The d_i processors for level i will be knowing about their work assigned to them by looking at the locations $e_{i-1}+1$ to e_i of *Alloc* in order of the processors(i.e 1 st processor looks at the location $e_{i-1}+1$, second processor looks at the location $e_{i-1}+2$ and so on.).

We indicate how a given processor j finds the work assigned to it. Suppose processor j is assigned to the location i of *Alloc*. Let us assume *Alloc*[i] contains q. Suppose the processor is assigned to the generation of k-combinations. The level in which we are operating is already known as the algorithm proceeds sequentially in terms of processing of levels. Recall that each k-combination's call has $r+1$ parts; though some parts may be null, but processors will always be assigned to useful parts only. Let us also assume that of the k-combinations the processor belongs to t th part of the k-combinations. We

would like to know what is the number of our processor out of the processors that are needed for the part t. This is $N[i]$. Once this is known, we can easily identify the work that should be done by processor j.

Main Processing

In the main processing, the algorithm proceeds bottom up. We first find the combinations of 1 elements to start the processing (base case of the recursive algorithm), then the combinations of 2 elements and so on. This processing finishes when we generate the relevant combinations of $\frac{n}{2}$ elements. Finally, we generate the r-combinations of n elements. This final step is done separately to get an optimal algorithm. Out of $\log n + 1$ levels, the first $\log n$ levels are similar. They correspond to the *for* loop in our algorithm presented. So we describe the processing for an arbitrary level.

Let us assume we are generating combinations for the i-th level; for i-th level, we will be generating combinations for 2^i elements. There will be two such sets of 2^i elements. More precisely we will be generating combinations for the first set $\{1, 2, \ldots, 2^i\}$ and the second set $\{2^i + 1, 2^i + 2, \ldots, 2^{i+1}\}$. For the sake of simplicity let us denote 2^i by w. So, we are generating combinations for first w elements and the next w elements. We are generating p-combinations for w elements (and the other set of w elements) for p varying from 1 to $min(w, r)$.

From Eq. (7.10), we can observe that to generate the p-combinations of w elements, we need the combinations from the two sets of elements, viz, the first set of $\frac{w}{2}$ elements and the second set of $\frac{w}{2}$ elements. Note that the lengths of the combinations that are needed from these two sets are same; The identity is symmetric in terms of first and second set of $\frac{w}{2}$ elements. The following observation is quiet useful in such situations where we need the same lengths of the combinations for different sets of elements of same size.

Observation: *Suppose we have with us the all the p-combinations of the elements* $\{1, 2, \ldots, n\}$, *then to generate the p-combinations of the set of elements* $\{b + 1, b + 2, \ldots, b + n\}$, *we add* b *to each element of each combination.*

Once we have the p-combinations of one set of elements the p-combinations of the adjacent set with the same size can be easily generated. One processor will be responsible for one element of a combination for the first set of w elements. So, that processor will

also add w to its value (which it is storing in its corresponding location) and stores that value in the corresponding location associated with the p-combinations for the other set of elements.

We now briefly describe how the processors are allocated and how they are doing their job. For each level, there are corresponding contiguous locations in the array *Alloc*. From the pre-processing we also know the number of processors that are needed for each level. For level i, we need $D[ir(r+1)] - D[(i-1)r(r+1)]$ processors. So we allocate that many processors for a given level and the processors identify their work by consulting their assigned location in *Alloc*. Processor j is associated with the location $j' = D[(i-1)r(r+1)] + j$ of *Alloc* for level i. Once the processor knows to which part of the p-combinations it is allocated i.e $Alloc[j']$, and its number with respect to the processors that are allocated for a given part i.e, $N[j']$, the processor copies the information from the previous combinations calculated for the last level of processing. Suppose the processor is allocated to a part: $C(\frac{w}{2}, k).C(\frac{w}{2}, p-k)$. Then $C(\frac{w}{2}, k).C(\frac{w}{2}, p-k).p$ processors are needed for this part and these processors copy the k-combinations from the first set of $\frac{w}{2}$ elements and $p-k$-combinations from the second set of $\frac{w}{2}$ elements. More precisely, we have to form p-combinations where the first k elements will be coming from the k-combinations of the first set of $\frac{w}{2}$ elements and the remaining $p-k$ elements will be coming from the $p-k$-combinations of the other set of $\frac{w}{2}$ elements. So, there will be $C(\frac{w}{2}, k).C(\frac{w}{2}, p-k)$ p-combinations in all for all the possibilities.

The locations where we store these p-combinations will be obtained from the array D. So, once a processor knows its number with respect to the part of the p-combinations, then it proceeds as follows. It first identifies the p-combination in which it is going to participate. Then it identifies whether it should retrieve an element from the k-combinations of first set of $\frac{w}{2}$ elements or the $p-k$-combinations of the second set of $\frac{w}{2}$ elements. Then it identifies the corresponding element and puts the element in the location corresponding to it in the p-combinations corresponding to first set of w elements and it adds w to its element and writes that in the position corresponding to it in the p-combinations for the adjacent set of w elements. The combination a processor is associated with, the element the processor should retrieve and where the processor

should store are easy to get, once we have the information obtained from pre-processing.

Regarding space allocation, we will have two four dimensional arrays. The first array will be for the first set of elements. The second array will be for the combinations corresponding to the adjacent set of elements. The first dimension of the array correspond to the level of bottom up processing. The second dimension corresponds to the length of combinations, the third dimension correspond to the number of combination and the fourth dimension corresponds to the actual combinations elements. As an example, the element that corresponds to $[4,3,5,2]$ is the 2nd element of the the 5 th combination of the 3-combinations of the set of elements $\{1,2,\ldots,2^4\}$, i.e $\{1,2,\ldots,16\}$.

Till now we have explained how we are generating combinations for all the levels except the top level. Once we have come to this stage this means that we have generated all needed combinations for the first set of $\frac{n}{2}$ elements and the second set of $\frac{n}{2}$ elements. We have to generate the r-combinations of set of n elements from these combinations. We use the identity (7.11) for processor allocation. The processor allocation is similar to the way we have shown earlier in our Generic processor allocation(see Section 2.3). Thus, we can get our r-combinations for the n elements.

We analyze the algorithm for time complexity and the work done.

We first consider the pre-processing part of the algorithm. Initially we are calculating the values of binomial coefficients. This calculation is implemented by finding factorials of $i!$ for $1 \leq i \leq n$ using prefix multiplication in $O(\log n)$ time with linear work[25]. Later we calculate the values of binomial coefficients in Step 4 in constant time with nr processors.

Then we do processor allocation. In Step 5 we formulated the array C which is useful for processor allocation. The array C store the information for all the recursive calls that will be made in algorithm. The array C is based on Eq. (7.10). There are $\log n - 1$ levels in all corresponding to the levels of recursive calls. Each level has got $min(p,r)$ calls in a level if we are generating combinations for p objects. But for simplicity the number of arrays we are filling here is r for each level. From Eq. (7.3), it follows that all the extra arrays and extra locations we are building up will contain zeroes. So, in the later processor allocation we won't be allocating processors to them. Thus, the array C is of length $(\log n - 1)r(r+1)$. Filling up C can be done in constant time with $O(\log n.r^2)$

```
Pre_Processing
1 for i = 1 to n in parallel do
      A[i] = i;
2 Prefix_mult(A);
A[0] = 1; // 0!  = 1

3 Calculate values of C(p,q) for 1 ≤ p ≤ n, 0 ≤ q ≤ r;
4 for q = 0 to r in parallel do
      for p = 1 to n in parallel do
          com[p,q] = A[p]/A[p-q].A[q];

Pre_Processing for Processor Allocation
5 Create an array C of length (log n − 1)r(r + 1). The locations
[(i − 1)r(r + 1)] + 1 to ir(r + 1) correspond to processor allocation
details for level i. If a level i starts at position START then the
processor allocation for generation of m-combinations out of 2^i
objects corresponds to locations START + [(m − 1)(r + 1) + 1] to
START+m(r+1), where 1 ≤ m ≤ r.
6 D = Prefix-sum(C);
7 for i = 1 to (log n − 1)r(r + 1) in parallel do
      start[i] = D[i-1]+1;
      size[i] = C[i];
      end[i] = start[i]+size[i]-1;
p = D[(log n-1)*r*(r+1)]
8 Generic_Processor_Allocation;
-----------------------------------------------------------
Algorithm Generate_Combinations(n, r)
Begin
   Generate 1-combinations of 1 object {1} and {2};
   for i = 1 to log n − 1 do
       Generate all the combinations related to set of 2^i
       objects and for an adjacent set of elements of the same size.
   Processor allocation details for the top level;
```

processors (Step 5).

In Step 6, we are calculating the prefix sum of array C into D. This takes a time of $O(\log(\log n.r^2))$ i.e $O(\log\log n + \log r)$ with a work of $O(\log n.r^2)$.

In Steps 7 and 8, we are getting the data structure for processor allocation. $Total$ is the final summation value in the prefix sums we have calculated above. In fact the $Total$ will be nothing but total number of operations that should be performed in the algorithm except the top level. We later show that this value $Total$ is bounded above by the expression $2.C(n,r).r$ (see Lemma 7.18).

Step 7 takes constant time with $O(\log n.r^2)$ processors. Generic processor allocation take a time of $O(\log\log *(Total + \log nr^2)))$ with $O(\log n.r^2 + Total)$ work on CREW PRAM (see Theorem 2.3). i.e time complexity is $O(\log\log *(n^{r+1})$, i.e $O(\log\log *n + \log\log *r)$.

Thus entire pre-processing takes a time of $O(\log n)$. The work done is $O(C(n,r).r + \log n.r^2 + nr)$ i.e, $O(C(n,r)r)$.

Lemma 7.18 *If $n = 2^k$ for some integer k and $1 \leq r \leq \frac{n}{2}$ then $\sum_{i=0}^{k-1}\sum_{j=0}^{min(2^i,r)} C(2^i,j).j \leq 2.C(n,r).r$*

Proof: The proof is by induction on k. We apply the basis when $n = 2$ and for all applicable values of r i.e, when $k = 1$.

Initially consider $n = 2$ and $r = 1$. It is obvious that

$C(1,0).0 + C(1,1).1 = 1 \leq 4 = 2.C(2,1).1$

Thus, basis is proved.

As part of induction hypothesis we assume that the lemma is valid for all n and for $1 \leq r \leq \frac{n}{2}$.

Let us consider the case when we are looking at $2n$ and for various values of r. We divide the proof into two cases. Note that $1 \leq r \leq n$ and $2n = 2^{k+1}$.

Let $LHS = \sum_{i=0}^{k}\sum_{j=0}^{min(2^i,r)} C(2^i,j).j$

Case $r \leq \frac{n}{2}$:.

So, $LHS = \sum_{i=0}^{k}\sum_{j=0}^{min(2^i,r)} C(2^i,j).j = \sum_{i=0}^{k-1}\sum_{j=0}^{min(2^i,r)} C(2^i,j).j + \sum_{j=0}^{min(2^k,r)} C(2^k,j).j$

In the above expression, consider the first part of the expression, for that all the assumptions of hypothesis are valid as n is a power of 2 and $r \geq 1$ and $r \leq \frac{n}{2}$ and hence by induction hypothesis,

$LHS \leq 2.C(n,r).r + \sum_{j=0}^{r} C(n,j).j$

From Eq. (7.11), $\sum_{j=0}^{r} C(n,j).j \leq C(2n,r).r$ as $r \leq n$. Hence,

$LHS \leq 2.C(n,r).r + C(2n,r).r$

Consider

$$C(2n,r)/C(n,r) = \frac{(2n)(2n-1).....(2n-r+1)}{(n)(n-1)(n-2)....(n-r+1)} \tag{7.13}$$

As $r \geq 1$, it follows that $C(2n,r)/C(n,r) \geq 2$, Thus, $C(2n,r) \geq 2.C(n,r)$.

So, $LHS \leq 2.C(n,r).r + C(2n,r).r \leq C(2n,r).r + C(2n,r).r = 2.C(2n,r).r$

Hence the lemma is valid for this case.

Case $r > \frac{n}{2}$:.

Recall that $r \leq n$.

Since $r > \frac{n}{2}$, it follows that

$LHS = \sum_{i=0}^{k} \sum_{j=0}^{min(2^i,r)} C(2^i,j).j = \sum_{i=0}^{k-1} \sum_{j=0}^{min(2^i,r)} C(2^i,j).j + \sum_{j=0}^{min(2^k,r)} C(2^k,j).j$.

$\sum_{i=0}^{k-1} \sum_{j=0}^{2^i} C(2^i,j).j + \sum_{j=0}^{r} C(2^k,j).j = \sum_{i=0}^{k-1} \sum_{j=0}^{min(2^i,\frac{n}{2})} C(2^i,j).j + \sum_{j=0}^{r} C(n,j).j$

since $r \leq n$, it follows that $min(n,r) = r$.

By using induction hypothesis, with $r = \frac{n}{2}$, it follows that

$LHS \leq 2.C(n,\frac{n}{2}).\frac{n}{2} + \sum_{j=0}^{r} C(n,j).j$

$2.C(n,\frac{n}{2}) \leq C(2n,\frac{n}{2})$ by Eq. (7.13).

From Eq. (7.5), $C(2n,\frac{n}{2}) \leq C(2n,r)$ as $r \leq n$ and $r > \frac{n}{2}$.

Since $\frac{n}{2} < r$ and using the above assertion,

$LHS \leq C(2n,r).r + \sum_{j=0}^{r} C(n,j).j$

From Eq. (7.11), the second term is less than or equal to $C(2n.r).r$.

Hence, $LHS \leq C(2n,r).r + C(2n,r).r = 2.C(2n,r).r$

As, the lemma is valid in this case also, the lemma is proved. ∎

We now take up the analysis of main processing. For the level i, we generate combinations of 2^i objects. We generate the p-combinations of these objects where $1 \leq p \leq min(2^i,r)$. More over we generate the combinations for two such adjacent sets.

Each level takes constant time. We count one operation as storing or obtaining an item of any combination. The generation of a single combination takes the number of operations equal to the length of the combination. So, the number of operations in level i are

$2[\sum_{j=0}^{min(r,2^i)} C(2^i, j).j]$

We have $\log n$ levels, excluding the top level. Hence the sum of operations of all these levels put together is

$2[\sum_{i=0}^{k-1} \sum_{j=0}^{min(r,2^i)} C(2^i, j).j]$ where $n = 2^k$.

From Lemma 7.18, the above sum is bounded above by $4.C(n,r).r$.

Since there are $\log n$ levels in all each level taking constant time. The time for this part is $O(\log n)$.

For top level, we allocate the processors by means of Generic processor allocation problem (see Section 2.3) using Eq. (7.11). This is similar to processor allocation in pre-processing.

So, processor allocation takes a time of $O(\log\log{*n} + \log\log{*r})$ time and the number of operations are $O(C(n,r).r)$. This step requires CREW PRAM.

Now, we consider the generation of r-combinations of n objects at the top level. This step takes constant time. The number of operations are $O(C(n,r).r)$.

We summarize in the following theorem.

Theorem 7.19 *The algorithm runs optimally with a time of $O(\log n)$ and the work is $O(C(n,r).r)$ on the CREW PRAM when $r \leq \frac{n}{2}$ and n is a power of 2.* ∎

7.4.2 Generalization of the Algorithm

In the earlier section, we have seen the algorithm to generate the r-combinations of n objects where n is a power of 2 and $r \leq \frac{n}{2}$. In this section, we will remove the first restriction of that algorithm *viz.* n is a power of 2. We will generate the r-combinations of n elements where $r \leq \frac{n}{2}$. The algorithm in this section heavily makes use of our earlier algorithm.

Like our earlier algorithm, this algorithm is also based on special cases of Vandermonde's convolution. We are interested in generating r-combinations of n elements. Let k is the maximum integer such that $2^k \leq n$. Since n is not a power of 2, $2^k < n$. From now onwards let us denote 2^k by n_1. In fact n_1 is the most significant bit in the binary representation of n.

Let $n_2 = n - n_1$. Observe that $n_1 \geq \lceil \frac{n}{2} \rceil$ and $n_2 \leq \lfloor \frac{n}{2} \rfloor$.

We would like to generate the r-combinations of n objects by first generating combinations for n_1 objects. From the combinations of n_1 objects $\{1, 2, \ldots, n_1\}$, we will get the combinations for n_2 objects $\{n_1 + 1, \ldots, n\}$. By combining together these combinations using Vandermonde's convolution, we get the r-combinations of n objects.

Depending on the value of r with respect to the value of n_2, we have two cases.

Case 1: $n_2 \geq r$

The following identity is a special case of Vandermonde's convolution Eq. (7.9) when $m_1 = n_1$ and $m_2 = n_2$.

$$\sum_{i=0}^{r} C(n_1, i).C(n_2, r-i) = C(n, r) \tag{7.14}$$

Thus, we need to generate the p-combinations for n_1 objects $\{1, 2, \ldots, n_1\}$ and n_2 objects $\{n_1 + 1, \ldots, n\}$ where $0 \leq p \leq r$.

The algorithm is given below.

1 Do Pre-processing and generate combinations for 1,2,...$n_1/2$ elements. Pre-processing involves solving processor allocation problem.

2 Do Processor allocation for generating p-combinations of n_1 elements $\{1, 2, ..., n_1\}$ where p varies from 1 to r;

3 Generate p-combinations of n_1 elements $\{1, 2, ..., n_1\}$ where p varies from 1 to r;

4 Generate p-combinations of n_1 elements $\{n_1 + 1, n_1 + 2, ..., 2n_1\}$ where p varies from 1 to r;

5 Assuming $n + 1, ..., 2n_1$ as dummy elements, rank the above combinations to obtain the p-combinations of n_2 elements $\{n_1 + 1, n_1 + 2, .., n\}$ where p varies from 1 to r;

6 Generate the r-combinations of n objects.

We initially do the pre-processing so that assuming we are generating the r-combinations of n_1 objects. Since n_1 is a power of 2, we will do this is in the same manner as we have done for the previous algorithm. Thus, we have got the p-combinations for m elements where m is a power of 2 and $m < n_1$ and p varies from 0 to $min(m, r)$.

We now do the processor allocation to generate p-combinations for n_1 elements $\{1, 2, \ldots, n_1\}$ where p varies from 1 to r. This processor allocation is based on the Eqs. (7.11) and (7.12) and using Generic processor allocation (see Section 2.3).

So, using the processor allocation we generate the p-combinations for n_1 elements $\{1, 2, \ldots, n_1\}$ where $1 \leq p \leq r$. Then, we generate the p-combinations for n_1 elements $\{n_1 + 1, n_1 + 2, \ldots, 2.n_1\}$ where p varies from 1 to r as follows. We copy the p-combinations for n_1 elements $\{1, 2, \ldots, n_1\}$ adding n_1 to each element of the combination.

We have to generate the p-combinations of n_2 elements $\{n_1 + 1, \ldots, n\}$ where $1 \leq p \leq r$ using the above combinations. Assume that the elements $\{n+1, n+2, \ldots, 2.n_1\}$ as *dummy* elements. We identify all those combinations which consist of at least one dummy element. The remaining combinations are just the p-combinations of the n_2 elements $\{n_1 + 1, \ldots, n\}$ where p varies from 1 to r. But to get these combinations together, we use *ranking*. We rank all the valid combinations using the procedure described in Section 7.3.1. At the end of this processing we have got all the needed p-combinations of n_2 elements $\{n_1 + 1, \ldots, n\}$ where p varies from 1 to r.

Once we have generated the p-combinations for n_1 elements $\{1, 2, \ldots, n_1\}$ and $\{n_1 + 1, \ldots, n\}$, we can easily generate the r-combinations of n objects $\{1, 2, \ldots, n\}$ using the identity (7.14).

We analyze the algorithm. Note that $n_2 \geq r$.

Lemma 7.20 *Step 1 takes a time of $O(\log n)$ with $O(C(n, r)r)$ work on CREW PRAM.*

Proof: In Step 1 we are doing the pre-processing for the processor allocation. The work for the pre-processing and the later generation of combinations is same. But the time is different; combination generation takes $O(\log n)$ time but pre-processing time depends on the work.

After the pre-processing is over, we generate the combinations for $1, 2, \ldots, n_1/2$ elements. The number of operations are

$$S = 2[\sum_{i=0}^{k-1} \sum_{j=0}^{min(2^i, r)} C(2^i, j).j] \tag{7.15}$$

where we assumed that $n_1 = 2^k$ for some integer k.

S can be bounded as follows. We consider two cases.

case 1: $r > n_1/2$

In this case S can also be written as follows.

$S = 2.[\sum_{i=0}^{k-1} \sum_{j=0}^{2^i} C(2^i, j).j]$

This can be bounded as $4.C(n_1, n_1/2).n_1/2$ (see Lemma 7.18). But since $C(n_1, n_1/2) \leq C(n, n_1/2) \leq C(n, r)$ (as $r \leq \frac{n}{2}$). Hence it follows that S can be bounded by $O(C(n, r).r)$.

case 2: $r \leq n_1/2$

S can be directly bounded by a direct application of Lemma 7.18. The S is bounded by $4.C(n_1, r).r$.

But since $C(n_1, r) \leq C(n, r)$, it follows that S is bounded by $O(C(n, r).r)$.

Thus, pre-processing has the cost $O(C(n, r).r)$. So, the time is $O(\log\log * (C(n, r).r))$ which is $O(\log\log * n + \log\log * r)$. The lemma follows. ∎

In Step 2 and 3, we are generating the p-combinations for n_1 elements where p varies from 1 to r. This takes a time of $O(\log\log * n + \log\log * r)$. This is the time taken for pre-processing. The generation of the combinations *per se*, takes constant time. The number of operations are

$\sum_{i=1}^{r} C(n_1, i).i$

From the Eq. (7.14), the above summation is bounded by $O(C(n, r).r)$.

In Step 4 we are generating the p-combinations for n_1 elements $\{n_1 + 1, \ldots, 2n_1\}$. This takes a time of $O(1)$ and the number of operations are same as above.

In Step 5, we are generating the p-combinations for n_2 elements by means of ranking where p varies from 0 to r. From Section 7.3.1, we know that ranking a p-combination takes a time of $O(\log p)$ and $O(p)$ work. Hence we generate the needed p-combinations in a time of $O(\log n)$ and with a work of $O(C(n, r).r)$.

Later we are generating the r-combinations of n elements. This is based on Eq. (7.14) and hence it takes a time of $O(\log\log *n + \log\log *r)$ (Note that we have to do processor allocation). The generation of the r-combinations per se, takes constant time with a work of $O(C(n,r).r)$. We summarize our above discussion in the following lemma.

Lemma 7.21 *The algorithm takes a time of $O(\log n)$ with optimal work $O(C(n,r).r)$ on the CREW PRAM with the condition that $n_2 \geq r$ and $r \leq \frac{n}{2}$.* ∎

Case 2: $n_2 < r$

In this case we aim at eliminating the zero terms from identity (7.14) to get

$$\sum_{i=0}^{n_2} C(n_1, r-i).C(n_2, i) = C(n,r) \tag{7.16}$$

Note that since $n_1 \geq \lceil \frac{n}{2} \rceil$ and $r \leq \lceil \frac{n}{2} \rceil$, it follows that $n_1 \geq r$. So, in other words we need to generate the p-combinations for n_1 objects $\{1, 2, \ldots, n_1\}$ where p varies from $r - n_2$ to r and p-combinations for n_2 objects $\{n_1 + 1, \ldots, n\}$ where p varies from 0 to n_2. The algorithm is given below.

1 Do Pre-processing and generate combinations for $1, 2, ..., n_1/2$ elements. Pre-processing involves solving processor allocation problem.

2 Do Processor allocation for generating p-combinations of n_1 elements $\{1, 2, ..., n_1\}$ where p varies from $r - n_2$ to r;

3 Generate p-combinations of n_1 elements $\{1, 2, ..., n_1\}$ where p varies from $r - n_2$ to r;

4 Generate p-combinations of n_1 elements $\{n_1 + 1, n_1 + 2, ..., 2n_1\}$ where p varies from 0 to n_2;

5 Assuming $n + 1, .., 2n_1$ as dummy elements, rank the above combinations to obtain the p-combinations of n_2 elements $\{n_1 + 1, n_1 + 2, .., n\}$ where p varies from 0 to n_2;

6 Generate the r-combinations of n objects.

We initially do the pre-processing assuming we are generating the r-combinations of n_1 objects. Since n_1 is a power of 2, the way we will do this is same as earlier we have done for the previous algorithm. Once this is done, we have got the p-combinations for m elements where m is a power of 2 and $m < n_1$ and p varies from 0 to $min(m, r)$.

We next do the processor allocation to generate p-combinations for n_1 elements $\{1, 2, \ldots, n_1\}$ where p varies from $r - n_2$ to r. This processor allocation is based on the Eqs. (7.11) and (7.12) with appropriate special cases of that identities and using Generic processor allocation (see Section 2.3). So, using the processor allocation we generate the p-combinations for n_1 elements $\{1, 2, \ldots, n_1\}$ where p varies from $r - n_2$ to r.

In a similar way, we generate the p-combinations for n_1 objects $\{1, 2, \ldots, n_1\}$ where p varies from 1 to n_2. From these we generate the p-combinations for n_1 objects $\{n_1 + 1 \ldots, 2.n_1\}$, by copying the p-combinations for n_1 elements $1, 2, \ldots, n_1$ adding n_1 to each element of the combination.

From these combinations, we generate the p-combinations of n_2 elements $\{n_1 + 1, \ldots, n\}$ where p varies from 1 to n_2. We assume that the elements $n + 1, n + 2, \ldots, 2.n_1$ as *dummy* elements. We identify all those combinations which consist of at least one dummy element. All the remaining combinations are just the p-combinations of the n_2 elements $\{n_1 + 1, \ldots, n\}$ where p varies from 0 to n_2. To get these needed combinations together, we use *ranking*. We rank all the valid combinations using the procedure described in Section 7.3.1. By the end of this processing we have got all needed p-combinations of n_2 elements $\{n_1 + 1, \ldots, n\}$ where p varies from 1 to n_2.

Once we have generated the p-combinations for n_1 elements $\{1, 2, \ldots, n_1\}$ where p varies from $r - n_2$ to r and p-combinations for n_2 objects $\{n_1 + 1, \ldots, n\}$ where p varies from 0 to n_2, we can easily generate the r-combinations of n objects $\{1, 2, \ldots, n\}$ using the identity (7.16).

We analyze the above algorithm noting that $n_2 < r$.

Step 1, takes a time of $O(\log n)$ with $O(C(n, r)r)$ work (see Lemma 7.20).

In Steps 2 and 3, we are generating the p-combinations for n_1 elements where p varies from $r - n_2$ to r. This takes a time of $O(\log\log *n + \log\log *r)$, the time taken for pre-processing. The generation of the combinations per se takes constant time. The number of operations are

$\sum_{i=r-n_2}^{r} C(n_1, i).i$

From Eq. (7.16), the above summation is bounded by $O(C(n,r).r)$.

In Step 4 we are generating the p-combinations for n_1 elements $\{n_1 + 1, \ldots, 2.n_1\}$. This takes a time of $O(\log\log *n + \log\log *r)$ (pre-processing for processor allocation) and the number of operations are:

$S = \sum_{i=0}^{n_2} C(n_1, i).i$

The following combinatorial identity is a special case of Eq. (7.9)

$\sum_{i=0}^{n_2} C(n_1, i).C(n_2, n_2 - i) = C(n, n_2)$

It follows that S is bounded by $O(C(n, n_2).n_2)$. But since $C(n, n_2) \leq C(n, r)$ (since $n_2 < r$ and $r \leq \frac{n}{2}$). S is bounded by $O(C(n,r).r)$.

In Step 5, we are generating the p-combinations for n_2 elements by means of ranking where p varies from 0 to n_2. From Section 3.2 we know that ranking a p-combination takes a time of $O(\log p)$ and $O(p)$ work. Hence we generate the needed p-combinations in a time of $O(\log n)$ and with a work of $O(C(n,r).r)$.

We then generate the r-combinations of n elements. This is based on Eq. (7.16) and it takes a time of $O(\log\log *n + \log\log *r)$ (we have to do processor allocation). The generation of the r-combinations per se takes constant time with a work of $O(C(n,r).r)$. We summarize our above discussion in the following lemma.

Lemma 7.22 *The algorithm takes a time of $O(\log n)$ with optimal work $O(C(n,r).r)$ on the CREW PRAM with the condition that $n_2 < r$ and $r \leq \frac{n}{2}$.* ∎

By combining our earlier two lemmas, we obtain the following theorem.

Theorem 7.23 *The algorithm can generate the r-combinations of n objects with a time complexity of $O(\log n)$ and optimal $O(C(n,r).r)$ work on the CREW PRAM when $r \leq \frac{n}{2}$.* ∎

7.4.3 CASE $r > \frac{n}{2}$

Now we are going to remove the last remaining restriction of earlier algorithm viz. $r \leq \frac{n}{2}$. The way we remove this is similar to we have done for our preceding algorithm.(see Section 7.3)

If $r > \frac{n}{2}$, then note that $n-r \leq \frac{n}{2}$. So we initially generate the $n-r$-combinations of n objects using the earlier algorithm described above. Later on for each $n-r$ combination, we generate its corresponding r-combination using prefix sums in a straight forward way.

Hence this leads to the following theorem (Note that n is $O(r)$ when $r > \frac{n}{2}$),

Theorem 7.24 *The algorithm will generate the r-combinations of n objects with a time complexity of $O(\log n)$ and optimal work $O(C(n,r).r)$ on CREW PRAM.* ∎

7.4.4 Generation of Combinations Lexicographically

The above algorithm does not generate the combinations in lexicographic order. But we can easily remedy this. We rank each combination to get its rank in lexicographic order. This can be done in $O(\log r)$ time with $O(r)$ work for each r-combination. So this leads to the following theorem.

Theorem 7.25 *The algorithm will generate the r-combinations of n objects in lexicographic order with a time complexity of $O(\log n)$ and optimal work $O(C(n,r).r)$ on CREW PRAM.*

Chapter 8

Conclusions

Our algorithms for the generation of combinatorial objects can be classified into two types—linear time algorithms and poly-logarithmic time algorithms. The linear time algorithms are derived using the construction of combinatorial trees. The concept of combinatorial tree we have defined, is very general and we believe its use extends beyond the present thesis. It can as well be a starting point for deriving new algorithms for similar combinatorial problems. The poly-logarithmic time algorithms are derived using divide-and-conquer paradigm. In general our algorithms are derived by giving algorithmic interpretation to well known combinatorial identities. This approach looks very promising for deriving parallel algorithms for combinatorial generation problems and needs to be explored further.

Note that our poly-logarithmic algorithms for generation of permutations and combinations are such that depending on the relative values of r and n, the r-recursive algorithm will be faster for some range and the n-recursive algorithm will be faster for the other range. It will be a good idea to design an algorithm which integrates the features of both these algorithms.

In the remaining part of this chapter, we will concentrate on the open problems in combinatorial generation. For the generation of permutations, we have looked at the basic problem of generating r-permutations of n objects. There are a number of variations and extensions to this problem for which no parallel algorithms are available on PRAM. They include the generation of: permutations with restricted repetitions, permutations

with unrestricted repetitions (variations), rosary permutations, cyclic permutations, alternate permutations, and permutations with fixed number of cycles (Stirling numbers). In the case of combinations also we have looked at the basic problem of generating r-combinations of n objects. The related open problems include the generation of: combinations with repetitions and gray codes.

There are a number of other problems in combinatorial generation which we have not touched upon and are awaiting parallel algorithms. They include the generation of: compositions with and without restrictions, and partitions with and without restrictions. Closely related to combinatorial generation are graphical enumeration problems. In this field also there are lot of problems for which no parallel algorithms are known. We believe our techniques can be applied to these problems because of their general nature.

Throughout the thesis, our concentration has been on the generation of combinatorial objects in *lexicographic order*. There is another well known order, *minimal change order*[16], which also has got important practical applications. It is a good idea to extend the algorithms to generate the combinatorial objects in this order. In general to generate the combinatorial objects in an orderly manner, *ranking* and *unranking* functions are needed. But for most of the problems, we mentioned above, the ranking and unranking functions are yet to be designed.

In the thesis, our emphasis has been on the generation of *all* combinatorial objects. An important variation of this approach is *uniform* generation of a *random* combinatorial object. This has also got large number of applications. For most of the problems mentioned above, no parallel algorithms are known for random generation of combinatorial objects.

Practically no non-trivial lower bounds for *time* are known for the generation of combinatorial objects. Hence it remains to be seen whether our algorithms can be improved (in terms of PRAM model or time or both) or they can be shown to be time optimal.

We showed that given permutations in lexicographic order, we can generate the derangements in lexicographic order. That is, getting improved algorithms for generating permutations has added advantage of automatically providing improved algorithms for

generating derangements. We have *not* shown that permutation generation and derangement generation are equivalent. It will be interesting to understand the relationship between these problems more deeply.

The approach we have presented for generating derangements from permutations can be cast in a much more broader perspective. Similar idea works in a situation where we are given set of combinatorial objects and we would like to obtain a subset of these elements which satisfy certain restrictions.

Bibliography

[1] AKL, S. G. A new algorithm for generating derangements. *BIT 20* (1980), 2–7.

[2] AKL, S. G. Adaptive and optimal parallel algorithms for enumerating permutations and combinations. *The Computer Journal 30*, 5 (1987), 433–436.

[3] AKL, S. G. *The Design and Analysis of Parallel Algorithms.* Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[4] AKL, S. G., GRIES, D., AND STOJEMENOVIC, I. An optimal parallel algorithm for generating combinations. *Information Processing Letters 33* (1989), 135–139.

[5] AKL, S. G., MEIJER, H., AND STOJMENOVIC, I. An optimal systolic algorithm for generating permutations in lexicographic order. *Journal of Parallel and Distributed Computing 20*, 1 (1994), 84–91.

[6] ANDERSON, G., AND MILLER, G. L. Parallel deterministic list ranking. *Algorithamica* (1988), 770–785.

[7] BELLMAN, R., COOKE, AND LOCKETT. *Graphs, Algorithms and Computers.* Academic Press, New York, 1970.

[8] BERKMAN, O. *Paradigms for Very Fast Parallel Algorithms.* PhD thesis, Dept. of CSE, Tel Aviv University, Tel Aviv, Israel, 1991.

[9] BHATT, P. C. P., DIKS, HAGERUP, T., RADZIK, PRASAD, V. C., AND SAXENA, S. Improved parallel deterministic integer sorting. *Information and Computation 94* (1991), 29–47.

[10] CHAN, B., AND AKL, S. G. Generating combinations in parallel. *BIT* (1986), 6–10.

[11] CHEN, G. H., AND CHERN, M. Parallel generation of permutations and combinations. *BIT 26*, 3 (1986), 277–283.

[12] COLE, R. Parallel merge sort. *SIAM Journal on Computing 17*, 4 (1988), 770–785.

[13] CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. *Introduction to Algorithms.* MIT Press, Cambridge, MA and McGraw-Hill, New York, 1990.

[14] DEO, N. *Graph Theory: With Applications to Computer Science and Engineering.* Prentice Hall, Englewood Cliffs, New Jersey, 1974.

[15] DJOKIC, B., MIYAKAWA, M., SEKIGUCHI, S., SEMBA, I., AND STOJMENOVIC, I. Parallel algorithms for generating subsets and partitions. In *Lec. Notes in Comp. Sc., SIGAL* (1990), vol. 450.

[16] EVEN, S. *Algorithmic Combinatorics.* Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[17] FORTUNE, S., AND WYLLIE, J. Parallelism in random access machines. In *Proc. of* 10^{th} *ACM Symp. on Theory of Comp.* (1978).

[18] GHOSH, R. K., MOONA, R., AND GUPTA, P. *Foundations of Parallel Processing.* Narosa, New Delhi, 1995.

[19] GIBBONS, A., AND RYTTER, W. *Efficient Parallel Algorithms.* Cambridge University Press, Cambridge, 1988.

[20] GRAHAM, R. L., KNUTH, D. E., AND PATASHNIK, O. *Concrete Mathematics: A Foundation for Computer Science.* Addison Wesley Publishing Co., Reading, Mass., 1989.

[21] GUPTA, P., AND BHATTACHARJEE, G. P. Parallel generation of lexicographic combinations. In *Conference Record of Foundations of software Technology and Theoretical Computer Science* (1981), pp. 193–200.

[22] GUPTA, P., AND BHATTACHARJEE, G. P. Parallel generation of permutations. *The Computer Journal 26*, 1 (January 1983), 97–105.

[23] GUPTA, P., AND BHATTACHARJEE, G. P. A parallel derangement generation algorithm. *BIT 29* (1989), 14–22.

[24] HAGERUP, T., AND SHEN, H. Improved non-conservative sequential and parallel integer sorting. *Information Processing Letters 36* (1990), 57–63.

[25] JAJA, J. *An Introduction to Parallel Algorithms.* Addison Wesley, Reading, Mass., 1992.

[26] KNUTH, D. E. *Fundamental Algorithms*, 2 ed., vol. 1, The Art of Computer Programming. Addison Wesley Publishing Co., Reading, Mass., 1973.

[27] KNUTH, D. E. *Sorting and Searching*, 2 ed., vol. 3, The Art of Computer Programming. Addison Wesley, Reading, Mass., 1973.

[28] LEHMER, D. H. The machine tools for combinatorics. In *Applied Combinatorial Mathematics*, E. F. Backenbach, Ed. John Wiley, 1964, ch. 1.

[29] LEIGHTON, F. T. *An Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Addison Wesley, Reading, Mass., 1992.

[30] LIU, C. L. *Introduction to Combinatorial Mathematics.* Mc GrawHill, New York, 1968.

[31] LOVASZ, L. *Combinatorial Problems and Exercises*, 2 ed. Elsevier Science, Amsterdam, 1993.

[32] MANBER, U. *Introduction to Algorithms: A Creative Approach.* Addison Wesley Publishing Co., Reading, Mass., 1989.

[33] NIJENHUIS, A., AND WILF, H. S. *Combinatorial Algorithms: for Computers and Calculators*, 2 ed. Academic Press, New York, 1978.

[34] RAJAN, V., GUPTA, P., AND GHOSH, R. K. Parallel generation of combinations and permutations. In *National Symposium in Theoretical Computer Science* (1991).

[35] RAJASEKARAN, S., AND REIF, J. H. Optimal and sub-logarithmic time randomized parallel sorting algorithms. *SIAM Jornal on Computing 18*, 3 (1989), 594–607.

[36] REINGOLD, E. M., NIEVERGELT, J., AND DEO, N. *Combinatorial Algorithms: Theory and Practice.* Prentice Hall International, Englewood Cliffs, New Jersey, 1977.

[37] RIORDAN, J. *An Introduction to Combinatorial Analysis.* John Wiley & Sons, 1958.

[38] SEDGEWICK, R. Permutation generation methods. *ACM Computing Surveys 19*, 2 (1977), 137–164.

[39] SPRINGSTEEL, F., AND STOJMENOVIC, I. Parallel general prefix computations with geometric, algebraic, and other applications. *International Journal of Parallel Programming 18*, 6 (1989), 485–503.

[40] SVED, M. Counting and recounting. *The Mathematical Intelligencer 5*, 4 (1983), 21–26.

[41] TSAY, J. C., AND LIN, C. J. A systolic design for generating combinations in lexicographic order. *Parallel Computing 26*, 2 (1992), 6–10.

[42] VELLEMAN, J. *How to Prove It.* Cambridge University Press, 1993.

[43] WILF, H. S. The "why don't you just ....?" barrier in discrete algorithms. *American Mathematical Monthly 86* (1979), 30–36.

CSE-1997-M-RAO-FAR

123434